COMP 542 Object Oriented Programming

Lecture Notes

Peter Grogono

March 2000

Department of Computer Science
Concordia University
Montreal, Quebec

COMP 542 Object Oriented Programming

Winter 2000

Contents

1 About the Course

1.1 Web Page

The web page for the course is `http://www.cs.concordia/~faculty/grogono/comp542.html`

1.2 Objective

When you have completed this course, you should be able to design, implement, test, and document an object oriented program in collaboration with other members of a team.

1.3 Assumptions

You have some previous experience of C++, from a course such as COMP 248, work experience, or in some other way.

1.4 Text Book

The recommended text for the course is:

- *A Practical Introduction to Software Design with C++*. Steven P. Reiss. John Wiley and Sons. ISBN 0–471–24213–6.
  Website: `http://tux.cs.brown.edu/people/spr/designbook/`.

Other useful books are listed on the web page.

1.5 Evaluation

- You will be evaluated on the basis of quizzes (15%), assignments (25%), and a programming project (60%).
- You will do the quizzes and assignments individually and the project as a member of a team.
- There will be about five assignments. The assignments will be primarily theoretical, since the project provides the practical component. Instructors will inform you of their policy with respect to late assignments.
- There will be two or three in-class quizzes, at the instructor's discretion; at least one quiz will be marked before the course drop date.

1.6 Computing Facilities

The laboratory for COMP 542 is H 929. This laboratory is equipped with fast PCs running Windows NT and Linux. Under NT, you can use Microsoft Developer Studio to build software in either C++ or Java. Under Linux, you can use the Gnu g++ compiler and other unix-like tools to build software.

1.7 Course Project

The programming project contributes 60% of your marks for the course (see above).

Each project will be completed by a team of about 5 students. Instructors will divide students into teams; the teams may make minor adjustments to their membership.

1.7.1 Suggested Schedule (Section E)

5	January	Instructor collects names
10	January	Instructor proposes teams
14	January	Teams submit names of team leader and team members

1.7.2 Roles of Team Members

One member of the team will be chosen by the team as its leader. The team leader acts as the communication channel between the instructor and the team. Apart from this, the responsibilities of the team leader are the same as for other members of the team.

Each team decides how to allocate its resources. One possibility would be for all of the members to collaborate in all of the tasks. At the other extreme, the team could allocate one person for design, one person for implementation, etc. In practice, something in between these extremes is probably best. For example, two team members could take responsibility for the design, but they would also consult other team members during the design process. Keep in mind the project deliverables when assigning roles to team members.

Disputes. Arguments within teams are common. If possible, the team should resolve disagreements by itself. If this doesn't work, the next step is to consult the tutor. If the tutor cannot solve the problem, the team leader should inform the instructor, who will probably arrange to meet with the whole team.

Credit. Another potential problem with team work is that the instructor must give equal credit to all team members. Team members may perceive this as unfair if some did more work than others. Problems of this kind should be solved by the team itself as far as possible. Individuals who feel that were treated unfairly should say so in their individual reports.

1.7.3 Project Deliverables

Each team must submit various deliverables as the project proceeds. The dates given here are tentative dates for Section E.

Proposal. (24 January)

The proposal is a brief (one or two page) description of the project that the team has decided to implement. Instructors will read the proposals carefully in order to decide whether they are appropriate for the course. They may suggest changes (either simplifications or extensions) of the proposed project or they may request a new proposal for a more appropriate project. Since it is best to avoid writing two proposals, you should consult the instructor before the due date if you have any doubts about the acceptability of your project.

Design. (14 February)

The design document consists of three components:

1. a general description of the software;
2. a high-level design showing the methods provided by each class and the interactions between classes; and
3. a detailed design showing the attributes of each class and describing how each of its methods will be implemented.

Instructors will review the designs and give their opinions to the proposing team. The purpose of the design review is mainly to avoid a situation in which a team gets stuck trying to implement an infeasible design.

Demonstration. (Starting 10 April)

Teams will demonstrate the programs they have written to the instructor. The purpose of the demonstration is to give the instructor an opportunity to see the program running, to try it out, and to discuss the project informally with the team. All team members must attend the demonstration.

Final Submission. (28 April)

To complete the project, each team must submit the following documents:

1. A *User Manual* explaining how to operate the program from a user's perspective.
2. *Design Documentation*, as described above, but possibly modified in the light of experience.
3. *Implementation Notes* describing particular features of the implementation, problems encountered and how they were solved, etc.
4. Complete source code.

In addition, you must submit an *individual* report of your own experience in working on the project: what you contributed, what was easy, what was hard, whether you had disagreements, etc.

For items 1 through 4 above, all members of a team will receive the same mark — up to 45% of the total course mark. Instructors will use the individual report to determine each team members's contribution to the project — up to 15% of the total course mark.

1.7.4 Choosing a Project

You have to design and implement an object oriented program written in C++. (Other object oriented languages, such as Java, may be acceptable, but you must obtain permission from the instructor before using a language other than C++.)

The program can be quite ambitious (it is worth 5 students $\times$ 4 credits $\times 60\% = 12$ credits) and should make extensive use of object oriented techniques. The objective of the project,

however, is a *high quality software product*: software that is user-friendly, robust, and well-documented. *Credit will be given for quality rather than quantity.*

Some ideas for projects are given on the web page
`www.cs.concordia.ca/~faculty/grogono/projects.html`.

2 Phases of Software Development

Software development is traditionally divided into various phases, which we describe briefly here. For small projects, the phases are carried out in the order shown; for larger projects, the phases are interleaved or even applied repetitively with gradually increasing refinement.

2.1 Requirements

The purpose of requirements analysis is to find out what the program is expected to do. This is done in the *application domain*. The result of requirements analysis might be that we don't need a program at all.

2.2 Specification

A specification is a precise description of what the program is supposed to do (but *not* how it works). The specification bridges the application domain and the *implementation domain* because it describes the external behaviour of the computer system. The specification can also be viewed as a *contract* between the client who wants the program and the software company that is supplying it.

2.3 Design

The design is a precise description of the internal structure of the program. The design of an object oriented program is usually a description of the classes and their interactions.

Design is the most important, but also most difficult, phase of software development. If the design is good, implementation should be straightforward. If the design is bad, the project may fail altogether.

2.4 Implementation

Implementation consists of writing the code: it is the activity that is often called "programming".

2.5 Testing

The code is tested thoroughly to ensure that the finished program meets its specification.

3 An Example of Design

As an example of object oriented design, we consider the *Solar System Simulator* described in Chapter 1 of Reiss's book. This program is an example of a *continuous simulation.*

3.1 Requirements

The program should simulate the motion of the sun, planets, satellites, and a few other assorted objects in the solar system.

The simulation should run faster than real time. For example, the earth might orbit the sun several times a minute rather than once a year.

The output of the simulation should be a graphical display of the simulated solar system with appropriate facilities for scaling, zooming, etc.

3.2 Specification

The program must provide a way of setting the initial conditions of the simulation: these conditions would consist of the mass, position, and velocity of each body at a particular time.

Starting from these initial conditions, the program uses Newton's laws to compute the position and velocity of each body at subsequent times. First-order approximations are sufficient.

(This is not a complete specification: we will discuss specifications in more detail later in the course.)

3.3 A Digression on Physics

Before proceeding with the design of the program, we will briefly review the underlying physics.

Consider two bodies in outer space, far from any other bodies. Suppose that their masses are m_1 and m_2 and that they are separated by a distance d. Then the gravitational force pulling them together is

$$F = \frac{G m_1 m_2}{d^2}$$

The force is actually a vector acting in a particular direction. We will indicate this by using a bold-face font for vector quantities.

For a collection of bodies, let $\mathbf{F}_{ij}$ be the gravitational force exerted on body i by body j. Then the total gravitational force acting on body i is

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{ij} \tag{1}$$

and for body i with mass m_i at position $\mathbf{p}_i$ Newton's second law gives

$$m_i \frac{d^2 \mathbf{p}_i}{dt^2} = \mathbf{F}_i$$

We can resolve this equation into components parallel to orthogonal axes XYZ. If we write x' for dx/dt and x'' for d^2x/dt^2, this gives

$$\begin{aligned} x'' &= F_x/m \\ y'' &= F_y/m \\ z'' &= F_z/m \end{aligned}$$

The specification says that we can use first-order approximation. To first order:

$$x'(t) = \frac{x(t+\Delta t) - x(t)}{\Delta t}$$

which we can write as

$$x(t+\Delta t) = x(t) + x'(t)\,\Delta t$$

and, similarly,

$$x'(t+\Delta t) = x'(t) + x''(t)\,\Delta t$$

3.4 Units

We could use astronomical units or arbitrary units. If we use astronomical units, we will have to deal with very large and very small numbers. To avoid this problem, we use arbitrary units. Specifically, we will start with a two-body system consisting of a sun s and a planet p in a circular orbit with

Mass of sun:	M_s	=	20
Mass of planet:	M_p	=	1
Radius of orbit:	r	=	20
Orbital velocity:	v	=	50

The gravitational force between the sun and the planet must balance the "centrifugal force" acting on the planet:

$$\frac{GM_sM_p}{r^2} = \frac{M_pv^2}{r}$$

From this equation, we can compute the gravitational constant, G in our arbitrary units:

$$\begin{aligned} G &= \frac{r^2}{M_sM_p} \times \frac{M_pv^2}{r} \\ &= \frac{v^2r}{M_s} \\ &= 2500 \end{aligned}$$

For the time increment, we set $\Delta t = 0.01$. The distance moved in one step is $v\Delta t = 0.5$ units. The length of the orbit is $2\pi r$ and the number of steps for an orbit is

$$\begin{aligned} \frac{2\pi r}{v\Delta t} &= 80\pi \\ &\approx 250 \end{aligned}$$

This is a reasonable compromise between speed and precision. To obtain greater precision, we could use a smaller value of Δt.

The values of v and r are chosen to give a circular orbit. To get an elliptical orbit, we can change one of them.

3.5 Subsystems

We can significantly reduce the amount of computation required by using subsystems.

Consider the system consisting of: the Sun, the Earth, the Moon, the planet Jupiter, and Jupiter's 11 major satellites. There are 15 bodies altogether and $15 \times 14/2 = 105$ interactions. In practice, the effect of Jupiter's satellites on the Earth is negligible and there is no point in computing it. Instead, we view this system as three *subsystems*:

1. the Sun;
2. the Earth and Moon;
3. Jupiter and 11 satellites.

Within this system, there are 3 interactions between the subsystems, 1 interaction within the Earth/Moon subsystem, and $12 \times 11/2 = 66$ interactions within Jupiter's subsystem, for a total of 70 interactions. This number can be reduced further if we ignore the effect of Jupiter's satellites on one another.

When we take the whole solar system into account (eight major planets of which all but Mercury and Venus have satellites) the saving achieved by subsystems is considerable.

3.6 Choosing Classes

The first step in object oriented design is to choose classes. In this example, it is fairly easy to choose the following classes:

Classes	Instances
Solar System	(unique)
Subsystem	Earth/Moon, etc.
Body	Sun, Earth, Moon, etc.

A class has *features*. A feature is either an *attribute* (or *data member* in C++) or a *method* (*function member* or *member function* in C++). We consider attributes first.

1. A Solar System has: a set of components, where each component is a subsystem or a body.
2. A Subsystem has: a set of components; a mass; a current position; and a current velocity.
3. A Body has: a mass; a current position; and a current velocity.

Notes:

- There is considerable similarity between these classes: we will discuss this later.
- The mass of a Subsystem is the total mass of its components.
- The position and velocity of a subsystem are actually the position and velocity of the centre of mass of its components.

- The position and velocity of a body that is a component of a subsystem are relative to the centre of mass of the subsystem.

For the first version of the design, we will ignore subsystems and consider single bodies only. Note that this is sufficient to simulate a solar system with a sun, planets, and moons, but the simulation will not be very efficient.

3.7 Methods

We will need a constructor for each object.

Following a convention for continuous simulations, we assume that each class has a method `update`. Given the position and velocity of the object at time t, this function computes the position and velocity at time $t + \Delta t$.

An object cannot update itself without additional information — the gravitational force acting on it. Consequently, each object will also need a method `setForce` to tell an object the total force acting on it. Alternatively, the objects could have two methods: `clearForce` to set the force to zero; and `addForce` to add a force to the current force.

The difference between these two approaches is that the first (`setForce`) requires that the forces on an object summed externally to the object while the second (`clearForce` and `addForce`) requires that the object itself sums the forces acting on it.

It is not obvious which approach is better, but we have to choose. This is a typical example of a *design choice* and good design consists mainly of *making the right choices*. Inevitably, we sometimes make the wrong choice. Then we either stumble on and end up with an inferior project, or we redesign the system with the right choice. Unfortunately, the second course is often infeasible (or too expensive).

Let us assume that we use the second approach (`clearForce` and `addForce`). Then the unique Solar System object first sends `clearForce` to all its components and then, for each pair of components i and j with $i \neq j$, sends `addForce(`$\mathbf{F}_{ij}$`)` to component i and `addForce(`$-\mathbf{F}_{ij}$`)` to component j. (This corresponds to equation (1) above.) Each subsystem treats its components analogously.

We will also need a way of finding out what is happening. We assume that this can be done by a method `display` that we send to an object: the object writes a message, or updates a graphical image, or something.

Acceleration, velocity, and position are all vector quantities. We could represent a vector simply as an array or we could introduce a class `Vector`.

The advantage of the class representation is that we can overload the standard operators to act on vectors. The result, for this application, is a considerable saving of writing. For example, to update a position, we write simply

```
pos += pos + vel * DT;
```

In this statement, we use `operator+=` and `operator+` defined for vectors and `operator*` to multiply a vector by a scalar.

3.8 Designing the class `Body`

We need classes for vectors and bodies. Vector is a "supplier" for body, but we do not know exactly which vector functions we will need until we have designed the body class.

We start by choosing the attributes (data members) for a body. We will need a scalar quantity for the mass and vector quantities for position, velocity, acceleration, and force. We can write:

```
class Body
{
public:
    . . . .
private:
    double mass;
    Vector pos;
    Vector vel;
    Vector acc;
    Vector force;
};
```

When a body is updated, the force and acceleration will be calculated and used to find new values of the velocity and position. It follows that the constructor should provide initial values for mass, position, and velocity, but can set acceleration and force to default values.

```
Body::Body (double iMass, Vector iPos, Vector iVel)
{
    mass = iMass;
    pos = iPos;
    vel = iVel;
    acc = Vector();
    force = Vector();
}
```

In writing the constructor, we assume that class `Vector` has an assignment operator and that the constructor `Vector()` constructs a zero vector.

The following method updates the state of a body during a time interval `DT` which corresponds to Δt above.

```
void Body::update (const double DT)
{
    acc = force / mass;
    vel += acc * DT;
    pos += vel * DT;
}
```

To implement this function, we must be able to multiply a vector by a scalar, divide a vector by a scalar, and the operator `+=` must be provided for vectors.

Next, we consider the updating method for the whole system. This method should probably not be a member function of class `Body` because it affects several bodies. Assume that we have an array `bods` of pointers to bodies and that it contains `numBodies` elements. The updating method is as follows.

```
void update(double DT)
{
    int a, b;
    for (a = 0; a < numBodies; a++)
        bods[a]->clearForce();
    for (a = 0; a < numBodies; a++)
        for (b = a + 1; b < numBodies; b++)
        {
            Vector sep = bods[a]->getPos() - bods[b]->getPos();
            Vector dir = sep.unit();
            double dist = sep.length();
            double mag = (GRAV * bods[a]->getMass() * bods[b]->getMass()) /
                (dist * dist);
            Vector force = dir * mag;
            bods[a]->addForce(-force);
            bods[b]->addForce(force);
        }
    for (a = 0; a < numBodies; a++)
        bods[a]->update(DT);
}
```

This code implies the need for some more methods for class `Body` (`clearForce`, `addForce`, `getMass`, and `getPos`) and some more methods for class `Vector` (binary $-$ and unary $-$, `unit`, `length`, and multiplication by a scalar). The new functions for class `Body` are shown below (`getMass` and `getPos` are omitted because they are trivial).

```
void Body::clearForce ()
{
    force = Vector();
}

void Body::addForce (Vector f)
{
    force += f;
}

void Body::update (const double DT)
{
    acc = force / mass;
    vel += acc * DT;
    pos += vel * DT;
}
```

3.9 Designing the class `Vector`

The class `Vector` is straightforward but makes use of a few C++ idioms that are worth noting. The representation is an array with three components:

```
class Vector
```

```
{
public:
    . . . .
private:
    double cpts[3];
};
```

There is a single constructor that initializes the three components but uses zero as the default value; this means that we do not need a separate constructor or method to create a zero vector:

```
Vector::Vector (double x = 0.0, double y = 0.0, double z = 0.0)
{
    cpts[0] = x;
    cpts[1] = y;
    cpts[2] = z;
}
```

We give clients access to the components by overloading the subscript operator, `[]`. In general, it is not good practice to expose the representation in this way. In the case of vectors, however, clients frequently need access to components for both reading and writing and it seems better to provide a concise notation rather than clumsy "get" and "set" functions.

```
double & Vector::operator[] (int i)
{
    return cpts[i];
}
```

The assignment operator is conventional, but note that we return a reference to the result. This is because C++ supports constructions of the form `a = b = c;` and because some programmers write things like `if(a = b)` (although this is considered poor practice nowadays). The same applies to `operator+=` although cascaded operations should be used rarely for this operator because they can be confusing.

```
Vector & Vector::operator= (Vector & other)
{
    cpts[0] = other[0];
    cpts[1] = other[1];
    cpts[2] = other[2];
    return *this;
}

Vector & Vector::operator+= (Vector & other)
{
    cpts[0] += other[0];
    cpts[1] += other[1];
    cpts[2] += other[2];
    return *this;
}
```

In C++, we can define operators for both binary minus (as in `u - v`) and unary minus (as in `-v`). We need both for the solar system simulation.

```
Vector Vector::operator- (Vector & other)
{
    return Vector
    (
        cpts[0] - other[0],
        cpts[1] - other[1],
        cpts[2] - other[2]
    );
}

Vector Vector::operator- ()
{
    return Vector (-cpts[0], -cpts[1], -cpts[2]);
}
```

We do not need any other methods that combine vectors, but we do need functions that multiply a vector by a scalar and divide a vector by a scalar. If these are defined as member functions of class `Vector` (as they are here), the vector must be the first argument. In other words, we must write `v * s` rather than `s * v`. We could avoid this restriction by defining them as non-member functions.

```
Vector Vector::operator* (double scalar)
{
    return Vector (cpts[0] * scalar, cpts[1] * scalar, cpts[2] * scalar);
}

Vector Vector::operator/ (double scalar)
{
    return Vector (cpts[0] / scalar, cpts[1] / scalar, cpts[2] / scalar);
}
```

The last two functions that we need are `length` and `unit`. In fact, we use `length` in the definition of `unit`.

```
double Vector::length ()
{
    return sqrt (sqr(cpts[0]) + sqr(cpts[1]) + sqr(cpts[2]));
}

Vector Vector::unit ()
{
    return (*this) / length();
}
```

Although we do not have an immediate use for it, a method that writes a vector to a stream is likely to be useful. (In fact, it *did* turn out to be useful — for debugging the simulation!)

These functions have a fairly standard form: note that the vector argument `v` is passed by reference (this is slightly more efficient) and the function *must* return a reference to the stream that it is given.

```
ostream & operator<< (ostream & os, Vector v)
{
    os << '(' << v[0] << ',' << v[1] << ',' << v[2] << ')';
    return os;
}
```

3.10 Discussion

The result of the design process is a *collection of classes*. For each class, we know:

1. the *interface* of the class — the set of methods that the class provides; and
2. the *implementation* of the class — the attributes of the class and how the methods affect them.

For the whole system, we also know how the classes interact: that is, what messages they send to one another and when they send them.

Distinguish carefully:

- A *class* is a static, compile-time entity.
- An *object* is a dynamic, run-time entity.
- There may be zero, one, or more *instances* of a class.
- The instances are *objects*.
- All instances of a class *behave* in the same way because they have the same methods.
- Each instance of a class has its *own*, *unique* data.

4 The Knight's Tour

This section is based on Chapter 2 of Reiss's book. The theme of this chapter is the design of an object oriented program that finds knight's tours. We begin with a precise description of the problem.

A *board* is a rectangular array divided into square cells. The *size* of a board is given by its length and width, in numbers of squares: an $m \times n$ board has mn squares.

Note: Reiss considers only *square* boards with $m = n$. However, there is no real need for this restriction.

A *knight* is a chess piece that moves on a board. When it is not moving, the knight is inside a particular cell. Each move takes it two steps in one direction and one step in an orthogonal direction. Since there are four choices for the first part of the move and two choices for the second part, there are eight possible moves from a given cell. In the table below, the initial position of the knight is labelled 0 and the cells that it can move to are labelled 1.

	1		1	
1				1
		0		
1				1
	1		1	

A *knight's tour* (KT) is a sequence of knight's moves with the following properties:

- the sequence starts and ends on the same square;
- the knight visits each other square on the board exactly once.

Programming Problem:

> Given a positive integer k, either find a knight's tour on a $k \times k$ board and display it, or print a message saying that the program could not find a tour. A tour should be printed as a $k \times k$ table of numbers $0, 1, \ldots, k^2 - 1$, in which 0 indicates the first and last position of the tour, 1 indicates the second position, and so on.

Before we start solving the program, we can make some observations about it.

- Since there are no knight's moves on a 2×2 board, there can be no KTs on a 2×2 board.
- Although a knight can move on a 3×3 board, it cannot move to or from the centre square. Consequently, there are no KTs on a 3×3 board.
- Consider a knight in a corner cell of a 4×4 board. Only two moves are possible and, in a KT, both must be taken. The same applies to the opposite corner. But this gives a closed path through only four cells (as shown in the table below). Consequently, there can be no KTs on a 4×4 board.

1			
		3	
	2		
			4

- Suppose that the cells are coloured alternately black and white, like a chess board. Then a move takes a knight either from a black cell to a white cell or from a white cell to a black cell. But a tour starts and ends on the same cell. It follows that a tour must visit an even number of cells and that there can be no KT on a $k \times k$ board if k is odd.

We conclude that KTs are possible only on $k \times k$ boards if k is even and $k \geq 6$. The program can immediately report "no tours" if k does not meet these conditions.

Finding a KT is a *search problem*. There are many knight's paths, but only some of them are KTs. When we start search, we do not even know whether a solution exists.

Many search problems, including the KT search, can be modelled as *trees*. The root of the tree corresponds to the start of the search (the knight's first position). Several moves from the start position are possible and these lead to first level nodes of the tree. From each of

these nodes, the second move leads to further nodes, and so on. The branching factor, or "bushiness", of the tree depends on how many moves are possible from a given position: for the knight, it varies from two to eight.

A move may bring the knight to a cell that it has visited before. Suppose that there are N steps in the path from the root of the tree to the node that corresponds to this cell. If $N = k^2$, we have found a KT. Otherwise we have failed, and there is no point in continuing this path because it visits a cell twice.

There are two principal ways of exploring a search tree:

- In *depth first order* (DFS) we follow a path until we either find a solution or get stuck. The name corresponds to the idea that we go as "deeply" into the tree as possible.
- In *breadth first search* (BFS) we visit all the vertexes at distance 1 from the root, then all the vertexes at distance 2 from the root, and so on.

We implement DFS by putting visited vertexes into a stack (LIFO) structure and BFS by putting visited vertexes into a queue (FIFO) structure.

- DFS is usually more efficient in *space* because only vertexes on the path are stored in the stack.
- BFS usually requires more space than DFS because the queue must contain all vertexes on the current path but usually contains other vertexes as well.
- If the tree contains infinite paths, DFS may fail to terminate but BFS will always find a solution if a solution exists.

Backtracking is a useful technique for exploring a search tree. We start at the root and build a path. The path contains *choice points*, which are the places where there are several possible extensions and we must choose between them. We continue making choices and building the path until we succeed or get stuck. If we get stuck, we move back to the most recent choice point and make a different choice. When we have tried all choices at a particular choice point, we move back further, and so on. If we reach a situation where we have backed up to the root and there are no further choices, there is no solution to the problem.

There are various ways of making choices:

- *Systematic* choices explore the paths in some organized way. For KTs, we could try knight's moves in the following order: up-left, left-up, left-down, and so on, going clockwise. If a move is impossible, because it leads to a previously-visited cell or off the board, we try the next one.
- *Random* choices use a random-number generator to explore the path.
- *Heuristic* choices use some "intelligent" way of making choices that is likely to find a solution more quickly than systematic or random search.

The search space for KTs looks large. The branching factor is between 2 and 8; if we assume an average of 4, then there are 4^{64} paths to explore on a 8×8 board. Since this is far too many for a systematic search, we will probably have to use a heuristic approach.

Since a KT visits every cell on the board, it doesn't seem to matter where we start. If we use a heuristic search, however, the starting point might make a difference: we would prefer to start a place where there were several options.

4.1 Design — Choosing the Classes

The first step in designing an object oriented program is to *find the classes*. Actually, we start by looking for *candidate classes* — entities that might become classes in the final program. There are various ways of finding classes. None of them is foolproof: the best guide is experience.

- Look for possible classes in the problem statement. Objects are things, and things are likely to be referred to as *nouns*. (It is also worth noting that a verb applied to a noun may suggest a method associated with an object.)
- You can rewrite the problem statement in your own way, being careful not to change its meaning in any significant way, and look for nouns in your own problem statement.
- Consider possible solutions and think about how they might work. A useful technique is to look for *scenarios* (also sometimes called *use cases*), which are examples of the way in which the proposed program may work.

In practice, looking for nouns tends to produce a large number of candidate classes (see, for example, Figure 2–3 on page 21 of Reiss's book). Consequently, we need efficient ways of eliminating classes from the list, or of not including them in the first place.

- Avoid classes that are too *simple*. If it looks as if the class would have only a single data member, reject it. If you can't think of any non-trivial methods for a class, consider rejecting it.
- Avoid classes that try to do *too much*. Nouns that refer to the entire solution or to large parts of it are probably not good classes.

The important nouns in the KT problem appear to be: `Knight`, `Board`, `Tour`, `Solution`, `Board`, and `Move`. However, it is not yet clear that all these will be classes in the finished program.

It is also important to realize that looking at the problem statement will not suggest *all* of the classes that may be needed for the solution. We may need "solution related" classes for the user interface, to represent data structures such as the search tree, to embody algorithms needed for the solution, and to "manage" the program's progress.

We should also consider parts of the solution that are likely to change. If possible, these parts should be put in separate classes so that they can be changed easily when the program is finished. The component of the KT solution that is least clear is the heuristic for building the search tree. This suggests that we introduce a class `Heuristic`; later on, if we want to try various heuristics, we can work within this class.

These considerations lead to an expanded list of candidate classes (Figure 2–6 on page 24): `Board`, `Direction`, `Heuristics`, `Knight`, `Move`, `Search`, `Solution`, `Square`, `Tour`, and `User Interface`.

4.2 The Top-Level Classes

Any system of sufficient complexity should have several top-level classes. A *top-level* class performs a major role in the solution and interacts with other top-level classes in significant

ways. Some systems have only one top-level class. A large system may have as many as ten top-level classes. It is probably wrong to have more than ten top-level classes because this suggests that we have not succeeded in breaking the solution down into manageable parts.

Top-level classes should be:

- *independent* — the functions of top-level classes should not overlap;
- *balanced* — the work done by each top-level class should be about the same; in particular, one class should not do most of the work;
- *interrelated* — the classes must interact to solve the problem; if they don't interact, there is something wrong with the solution.

Here are possible top-level classes for KT together with their main functions (Figure 2–7 on page 26):

`Main`. This class creates instances of other classes as needed; manage input and output via the user interface; find the tour.

`Board`. This is a "container" class that holds all of the cells of the board; it also manages the search and, in particular, the square in which the tour starts.

`Square`. An instance represents a square on the board; maintains a list of squares that can be reached in one knight's move from this square; and assists in the search.

`Heuristic`. This class contains algorithms that order the valid moves from a square so that the moves that are most likely to contribute to a solution come first.

`Solution`. This class stores partial solutions as the program runs and the complete solution if the program finds a KT.

The list of top-level classes is shorter than the list of candidate classes. To ensure that the design is complete, we review the list of candidate classes, checking that the functionality that they were supposed to provide is in fact included in the top-level classes (Figure 2–8 on page 27). Here are the candidate classes that are not included in the final design:

`Direction`. A direction is a (trivial) component of a `Move`.

`Knight`. A `Square` represents the current position of the imaginary travelling knight.

`Move`. A `Square` contains the set of valid moves. (This does not itself rule out the idea of a class `Move`. However, the amount of information contained in a move is small and does not warrant an extra class.)

`Search`. Searching is performed by the `Board` class.

`Tour`. This is essentially the same as the `Solution` class that we have included.

`User Interface`. The class `Main` provides user interface services. However, we might decide later that the user interface is sufficiently important to be a separate class.

4.3 Documenting the Design

Designs are documented using *text*, *diagrams*, or — usually — both. The hard part is to provide the right amount of information: too much information, and the design becomes an implementation; too little information, and the design cannot be implemented.

Reiss provides diagrams (Figures 2–9 and 2–10 on page 29). To complement this, we will provide a textual design that uses C++ (or Java) syntax. For each class, we write a skeleton declaration that shows the important methods and attributes.

```
class Main
{
public:
    void process();
private:
    the_board
    int board_size;
};

class Board
{
public:
    void setup ();
    void findTour ();
private:
    squares
};

class Square
{
public:
    void setup ();
    void findRestOfTour ();
private:
    int row_number;
    int column_number;
    legal_moves
};

class Heuristic
{
public:
    void orderMoves ();
private:
};

class Solution
{
public:
```

```
    void showSolution ();
private:
};
```

Note:

- The textual design lacks some of the information of the diagrammatic design. In particular, the text does not show interactions between classes.

  The diagrams do show interaction between classes but they do not show which methods implement these interactions. If we annotated the class declarations to show how methods made use of other classes, the declarations would be more informative than the diagrams.

- The class declarations contain different levels of detail. When we are fairly confident about something, we include details. For example, it seems reasonable to assume that the board size, row numbers, and column numbers can be represented as integers, and we have declared them as such. On the other hand, it is not yet clear how we should represent squares or legal moves, and these data names have been left without types.

- It is also not clear what data the classes `Heuristic` and `Solution` will need. Accordingly, we leave the `private` parts of their declarations blank.

4.4 Detailed Design

The final step in design is to provide sufficient information about the intended implementation to enable programmers to write the code. One way to do this is to write *pseudocode* for each method.

Pseudocode is a tricky concept because it must be precise about some things yet may be vague about others.

- Assume that the system will be implemented by several programmers. Each programmer is responsible for one or more classes. Programmers do not need to be told precisely how to implement a class, because they are capable of making decisions about representations. For example, we can trust them to decide between an array and a linked list. Thus the pseudocode can be vague about information *within a class*.

- The designer must assume that programmers will not communicate while writing their classes (although, in practice, they may). Consequently, all information about class interaction must be expressed *precisely and completely* in the pseudocode.

- Pseudocode is *not* source code. Pseudocode is not written in C++, Java, or Smalltalk. However, the designer usually knows what the implementation language will be, and can use pseudocode conventions that reflect the target language. When designing for C++ code, for example, we use C++ declaration style and control structures.

Pseudocode should be somewhat "higher level" than source code. For example, Reiss writes

```
for row = 0 to board_size - 1
....
Next row
```

rather than

```
for (int row = 0; row < board_size; row++)
....
Next row
```

The first version (pseudocode) is more readable, but there is a risk of mistranslation. In this case, the risk is small, because the pseudocode follows the C++ convention of starting a loop with index 0 and ending it at $N - 1$, where N is some natural "maximum". Pseudocode such as

```
for row = 1 to size
....
Next row
```

would not be appropriate in a C++ design because it would leave the implementor wondering whether to implement what was written (strictly, this is the correct approach) or translating it into the C++ idiom.

[I use the first version above, with the C++ bounds, but omit the line "`Next row`". P.G.]

The method `Main::process` controls the whole program.

```
void Main::process (int argc, const char **argv)
{
    board_size = Main::getBoardSize(argc, argv);
    the_board = new Board;
    set up board by calling the_board->Board::setup(board_size);
    if (tour exists for board size)
        result_tour = Board::findTour();
    else
        result_tour = no_tour;
    Main::output(result_tour);
}
```

Notes:

- The level of detail varies: when there is no further decisions to make, we write (almost) C++.
- Interactions between classes are already established precisely.
- The design seems awkward in the following sense. There are two possible outcomes: a tour, or a failure to find a tour. The program discovers the actual outcome in the statement

  ```
  if (tour exists for board size)
  ```

  and, if there is a tour, obtains it in the statement

  ```
  result_tour = Board::findTour();
  ```

This seems backwards, because we don't know if there is a tour until we have called `Board::findTour()` to find it!

Moreover, there is an `if` statement in the code for `Main::process()` that distinguishes the two cases and stores the result in `result_tour`. There is presumably another `if` statement in `Main::output()` that distinguishes between a "real tour" and a "`no_tour`".

We could avoid these problems like this:

```
....
Board::findTour();
if (Board::tourFound())
    Main::output(Board::getTour());
else
    report failure
```

An alternative interpretation of "`tour exists for board size`" is "it is possible that tours exist for this board size". In other words, the point of this test is to avoid calling `findTour` if we know without further analysis that no solution is possible. From previous reasoning, we know that there are no solutions if $k \leq 6$ or k is odd. Reiss's implementation of this method (page 483) seems to confirm this interpretation.

The point to notice here is that *the pseudocode was ambiguous* and that this led to a misunderstanding. Thus the example illustrates one of the weaknesses of pseudocode: its vagueness may lead to implementations that do not match the design.

```
Solution Board::findTour ()
{
    Square start = squares[1][2];
    Square sq0 = squares[0][0];
    start->Square::markUsed(sq0);
    sq0->Square::markUsed(start);
    sol = new Solution;
    sol->Solution::setup(board_size);
    start->Square::findRest(sol);
    return sol;
}
```

Since the tour must start and end at the same cell, we must treat the first step of the tour as a special case. We do this by assuming that the tour starts by going from $(0, 0)$ to $(1, 2)$ and marking these squares accordingly.

4.5 An Alternative Approach

The motivation for the KT problem in Reiss's book is to illustrate object oriented design. If we really wanted to solve the problem, instead of merely illustrating object oriented design techniques, we might look for other approaches.

Abstraction is a useful technique in problem solving. In the KT problem, we can abstract away from the board and its squares and, instead, think of the *graph* defined by knight's moves. In the undirected graph, each vertex corresponds to a cell and each edge corresponds

to a knight's move. This abstraction transforms the problem into the standard graph-theoretic problem of *finding a Hamiltonian circuit*. We can apply known results from graph theory to help us find solutions or identify situations in which there are no solutions. For example, a graph has a Hamiltonian circuit only if all of its vertexes have even degree.

The graph corresponding to a 4×4 board has 16 vertices. Suppose we number them like this:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

The following table shows the graph of knight's moves, expressed as edge lists. We can see immediately that no KT is possible, because vertices 2, 3, 5, 8, 9, 12, 14 and 15 (that is, the vertices on the edges between the corners) have odd degree.

1	7	10		
2	8	9	11	
3	5	10	12	
4	6	11		
5	3	11	14	
6	4	12	13	15
7	1	9	14	16
8	2	10	15	
9	2	7	15	
10	1	3	8	16
11	2	4	5	13
12	3	6	14	
13	6	11		
14	5	7	12	
15	6	8	9	
16	7	10		

4.6 Documenting the Design

There are various ways in which we can document a design. The following list shows a typical set of design documents.

Introduction. A brief description of what the program does and how it works. The introduction may also describe the other design documents to help the reader navigate them.

Software Architecture. A description of the system at a high level of abstraction. Alternative names are **Architectural Design** and **High Level Design**. For an object oriented program, the architecture is usually a collection of classes.

Diagrams show interactions between classes. It is important to distinguish the client/supplier, or "uses" relationship and the "inherits" relationship. This can be done either by using

two different kinds of arrow to connect classes in a single diagram, or by using separate diagrams for "uses" and "inherits".

In the diagram, a class appears as a rectangle. The rectangle should include the names of the class, names of its methods, and names of its attributes. If the class has many methods and attributes, list the important ones only.

In addition to the diagrams, the high level design should include a text description of each class. The text includes the name of the class and its important features. If a feature name is not self-explanatory, write a brief comment that explains its function. The text descriptions of classes are needed as well as the diagrams because there is space to include these comments.

Message Traces. Message trace diagrams (also called "use case diagrams") show how messages propagate in the system. Without them, it is hard to understand how the classes work together.

Detailed Design. The detailed design describes the role of each method and attribute of each class. If a method is simple, it can be described in a sentence or two. For more complicated methods, pseudocode may be better. If it is important to use a particular data structure, the detailed design should say so.

When describing classes, ensure that class interactions are complete and accurate: a programmer should be able to implement a class from the detailed design without consulting programmers working on other classes. The finer details of how the class itself works can be left to the implementor.

5 Coding Conventions

When the design has been completed, coding begins. Code must be written in such a way that people can read it. Compare

```
void calc (double m[],char *g){
double tm=0.0; for (int t=0;t<MAX_TASKS;t++) tm+=m[t];
int i=int(floor(12.0*(tm-MIN)/(MAX-MIN))); strcpy(g,let[i]);}
```

and

```
void calculateGrade (double marks[], char *grade)
{
    double totalMark = 0.0;
    for (int task = 0; task < MAX_TASKS; task++)
        totalMark += marks[task];
    int gradeIndex = int(floor(12.0 * (totalMark - minMarks) / (maxMarks - minMarks)));
    strcpy(grade, letterGrades[gradeIndex]);
}
```

These two function definitions are identical as far as the compiler is concerned. For a human reader, however, the difference is like night and day. Although a programmer can understand what `calc()` is doing in a technical sense, the function has very little "meaning" to a person.

The second version of the function differs in only two ways from the first: it makes use of "white space" (tabs, line breaks, and blanks) to format the code, and it uses descriptive identifiers. With these changes, however, we can quite easily guess what the program is doing, and might even find mistakes in it without any further documentation.

In practice, we would include comments even in a function as simple as this. But note how the choice of identifiers reduces the need for comments.

The three basic rules of coding are:

- Use a clear and consistent layout.
- Choose descriptive and mnemonic names for constants, types, variables, and functions.
- Use comments when the meaning of the code by itself is not completely obvious and unambiguous.

These are generalities; in the following sections we look at specific issues.

5.1 Layout

The most important rule for layout is that *code must be indented according to its nesting level.* The body of a function must be indented with respect to its header; the body of a `for`, `while`, or `switch` statement must be indented with respect to its first line; and similarly for `if` statements and other nested structures.

You can choose the amount of indentation but you should be consistent. A default `tab` character (eight spaces) is too much: three or four spaces is sufficient. Most editors and programming environments allow you to set the width of a tab character appropriately.

Bad indentation makes a program harder to read and can also be a source of obscure bugs. A programmer reading

```
while (*p)
    p->processChar();
    p++;
```

assumes that `p++` is part of the loop. In fact, it isn't, and this loop would cause the program to "freeze" unaccountably.

Although indentation is essential, there is some freedom in placement of opening and closing braces. Most experienced C++ programmers position the braces for maximum visibility, as in this example:

```
Entry *addEntry (Entry * & root, char *name)
    // Add a name to the binary search tree of file descriptors.
{
    if (root == NULL)
    {
        root = new Entry(name);
        if (root == NULL)
            giveUp("No space for new entry", "");
        return root;
```

```
        }
        else
        {
            int cmp = strcmp(name, root->getName());
            if (cmp < 0)
                return addEntry(root->left, name);
            else if (cmp > 0)
                return addEntry(root->right, name);
            else
                // No action needed for duplicate entries.
                return root;
        }
    }
```

Other programmers prefer to reduce the number of lines by moving the opening braces ("{") to the previous line, like this:

```
    Entry *addEntry (Entry * & root, char *name) {
        // Add a name to the binary search tree of file descriptors.
        if (root == NULL) {
            root = new Entry(name);
            if (root == NULL)
                giveUp("No space for new entry", "");
            return root;
        } else {
            int cmp = strcmp(name, root->getName());
            if (cmp < 0)
                return addEntry(root->left, name);
            else if (cmp > 0)
                return addEntry(root->right, name);
            else
                // No action needed for duplicate entries.
                return root;
        }
    }
```

The amount of paper that you save by writing in this way is minimal and, on the downside, it is much harder to find corresponding pairs of braces.

In addition to indentation, you should use blank lines to separate parts of the code. Here are some places where a blank line is often a good idea but is not essential:

- between method declarations in a class declaration;
- between variable declarations in a class declaration;
- between major sections of a complicated function.

Here are some places where some white space is essential. Put one or more blank lines between:

- `public`, `protected`, and `private` sections of a class declaration;

- class declarations (if you have more than one class declaration in a file, which is not usually a good idea);
- function and method declarations.

5.2 Naming Conventions

Various kinds of name occur within a program:

- constants;
- types;
- local variables;
- attributes (data members);
- functions;
- methods (member functions).

It is easier to understand a program if you can guess the kind of a name without having to look for its declaration which may be far away or even in a different file. There are various conventions for names. You can use:

- one of the conventions described here;
- your own convention;
- your employer's convention.

You may not have an option: some employers require their programmers to follow the company style even if it is not a good style.

As a general rule, *the length of a name should depend on its scope.* Names that are used pervasively in a program, such as global constants, must have long descriptive names. A name that has a small scope, such as the index variable of a one-line `for` statement, can be short: one letter is often sufficient.

Another rule that is used by almost all C++ programmers is that *constant names are written with upper case letters and may include underscores.*

5.2.1 Brown University Conventions

Appendix A of Reiss's book *Software Design* describes the coding standards used at Brown University for introductory software engineering courses. Most of the conventions are reasonable, but you can invent your own variations. Here are some of the key points of the "Brown Standard":

- File names use lower case characters only.

  UNIX systems distinguish cases in file names: `mailbox.h` and `MailBox.h` are different files. One way to avoid mistakes is to lower case letters only in file names.

Windows does *not* distinguish letter case in file names. This can cause problems when you move source code from one system to another. If you use lower case letters consistently, you should not have too many problems moving code between systems. Note, however, that some Windows programs generate default extensions that have upper case letters!

- Types and classes start with the project name.

  An abbreviated project name is allowed. For example, if your project is called `MagicMysteryTour`, you could abbreviate it to `MMT` and use this string as a prefix to all type and class names: `MMTInteger`, `MMTUserInterface`, and so on.

  [I don't think this is necessary for projects. The components of a project are usually contained within a single directory, or tree of directories, and this is sufficient indication of ownership. The situation is different for a library, because it must be possible to import library components without name collisions. P.G.]

- Methods start with a lower case letter and use upper case letters to separate words. Examples: `getScore`, `isLunchTime`.

  [I use this notation for both methods `and` attributes (see below). In the code, you can usually distinguish methods and attributes because method names are followed by parentheses. P.G.]

- Attributes start with a lower case letter and use underscores to separate words. Examples: `start_time`, `current_task`.

- Class constants use upper case letters with underscores between words. Examples: `MAXIMUM_TEMPERATURE`, `MAIN_WINDOW_WIDTH`.

- Global names are prefixed with the project name. Example: `MMTstandardDeviation`.

  [I use the same convention for methods and global functions. When a method is used in its own class, this can lead to ambiguity. In many cases, however, you can recognize methods because they appear as `o.method()` or `p->method()`. My rule for global variables is *avoid them* P.G.]

- Local variables are written entirely in lower case without underscore. Examples: `index`, `nextitem`.

Whatever convention you use, it is helpful to distinguish attributes from other variables. In *Large-Scale C++ Software Design*, John Lakos suggests prefixing all attributes with `d_`. This has several advantages; one of them is that it becomes easy to write constructors without having to invent silly variations. For example:

```
Clock::Clock (int hours, int minutes, int seconds)
{
    d_hours = hours;
    d_minutes = minutes;
    d_seconds = seconds;
}
```

5.2.2 Hungarian Notation

Hungarian notation was introduced at Microsoft during the development of OS/2. It is called "Hungarian" because its inventor, Charles Simonyi, is Hungarian. Also, identifiers that use this convention are hard to pronounce, like Hungarian words (if you are not Hungarian, that is).

If you do any programming in the Windows environment, you will find it almost essential to learn Hungarian notation. Even if you have no intention of ever doing any Windows programming, you should consider adopting this convention, because it is quite effective.

Hungarian variable names start with a small number of lower case letters that identify the type of the variable. These letters are followed by a descriptive name that uses an upper case letter at the beginning of each word.

For example, a Windows programmer knows that the variable `lpszMessage` contains a long p to a string terminated with a zero byte. The name suggests that the string contains a message of some kind.

The following table shows some commonly used Hungarian prefixes.

`c`	character
`by`	unsigned `char` or byte
`n`	short integer (usually 16 bits)
`i`	integer (usually 32 bits)
`x, y`	integer coordinate
`cx, cy`	integer used as length ("count") in X or Y direction
`b`	boolean
`f`	flag (equivalent to boolean)
`w`	word (unsigned short integer)
`l`	long integer
`dw`	double word (unsigned long integer)
`fn`	function
`s`	string
`sz`	zero-terminated string
`h`	handle (for Windows programming)
`p`	pointer

5.3 Commenting Conventions

Comments are an essential part of a program but you should not over use them. The following rule will help you to avoid comments:

> *Comments should not provide information that can be easily inferred from the code.*

There are two ways of applying this rule.

- Use it to eliminate pointless comments:

```
counter++;   // Increment counter.
```

```
// Loop through all values of index.
for (index = 0; index < MAXINDEX; index++)
{ .... }
```

- Use it to improve existing code. When there is a comment in your code that is not pointless, ask yourself "How can I change the code so that this comment becomes pointless?"

  You can often improve variable declarations by applying this rule. Change

```
int np;       // Number of pixels counted.
int flag;     // 1 if there is more input, otherwise 0.
int state;    // 0 = closed, 1 = ajar, 2 = open.
double xcm;   // X-coordinate of centre of mass.
double ycm;   // Y-coordinate of centre of mass.
```

  to

```
int pixelCount;
bool moreInput;
enum { CLOSED, AJAR, OPEN } doorStatus;
Point massCentre;
```

There should usually be a comment of some kind at the following places:

- At the beginning of each file (header or implementation), there should be a comment giving the project that the file belongs to and the purpose of this file.
- Each class declaration should have a comment explaining what the class is for. It is often better to describe an *instance* rather than the class itself:

```
class ChessPiece
{
    // An instance of this class is a chess piece that has a position,
    // has a colour, and can move in a particular way.
....
```

- Each method or function should have comments explaining what it does and how it works.

  In many cases, it is best to put a comment explaining *what the function does* in the header file or class declaration and a different comment explaining *how the function works* with the implementation. This is because the header file will often be read by a client but the implementation file should be read only by the owner. For example:

```
class MathLibrary
{
public:
    // Return square root of x.
    double sqrt (double x);
    ....
```

```
};

MathLibrary::sqrt (double x)
{
    // Use Newton-Raphson method to compute square root of x.
    ....
}
```

- Each constant and variable should have a comment explaining its role *unless its name makes its purpose obvious.*

5.3.1 Block Comments

You can use "block comments" at significant places in the code: for example, at the beginning of a file. Here is an example of a block comment:

```
/****************************************/
/*                                      */
/*       The Orbiting Menagerie         */
/*       Author:  Peter Grogono         */
/*       Date:      24 May 1984         */
/*         (c) Peter Grogono            */
/*                                      */
/****************************************/
```

Problems with block comments include:

- they take time to write;
- they take time to maintain;
- they become ugly if they are not maintained properly (alignment is lost, etc.);
- they take up a lot of space.

Nevertheless, if you like block comments and have the patience to write them, they effectively highlight important divisions in the code.

5.3.2 Inline Comments

"Inline comments" are comments that are interleaved with code in the body of a method or function. Use inline comments whenever the logic of the function is not obvious. Inline comments *must respect the alignment of the code.* You can align them to the current left margin or to one indent with respect to the current left margin. Comments must always confirm or strengthen the logical structure of the code; they must never weaken it.

Here is an example of a function with inline comments.

```
void collide (Ball *a, Ball *b, double time)
{
    // Process a collision between two balls.

    // Local time increment suitable for ball impacts.
    double DT = PI / (STEPS * OM_BALL);

    // Move balls to their positions at time of impact.
    a->pos += a->vel * a->impactTime;
    b->pos += b->vel * a->impactTime;

    // Loop while balls are in contact.
    int steps = 0;
    while (true)
    {

        // Compute separation between balls and force separating them.
        Vector sep = a->pos - b->pos;
        double force = (DIA - sep.norm()) * BALL_FORCE;
        Vector separationForce;
        if (force > 0.0)
        {
            Vector normalForce = sep.normalize() * force;

            // Find relative velocity at impact point and deduce tangential force.
            Vector aVel = a->vel - a->spinVel * (sep * 0.5);
            Vector bVel = b->vel + b->spinVel * (sep * 0.5);
            Vector rVel = aVel - bVel;
            Vector tangentForce = rVel.normalize() * (BALL_BALL_FRICTION * force);
            separationForce = normalForce + tangentForce;
        }

        // Find forces due to table.
        Vector aTableForce = a->ballTableForce();
        Vector bTableForce = b->ballTableForce();
        if (    separationForce.iszero() &&
                aTableForce.iszero() &&
                bTableForce.iszero() && steps > 0)
            // No forces: collision has ended.
            break;

        // Effect of forces on ball a.
        a->acc = (separationForce + aTableForce) / BALL_MASS;
        a->vel += a->acc * DT;
        a->pos += a->vel * DT;
        a->spin_acc = ((sep * 0.5) * separationForce + bottom * aTableForce) / MOM_INT;
        a->spinVel += a->spin_acc * DT;
        a->updateSpin(DT);
```

```
            // Effect of forces on ball b.
            b->acc = (- separationForce + bTableForce) / BALL_MASS;
            b->vel += b->acc * DT;
            b->pos += b->vel * DT;
            b->spin_acc = ((sep * 0.5) * separationForce + bottom * bTableForce) / MOM_INT;
            b->spinVel += b->spin_acc * DT;
            b->updateSpin(DT);

            steps++;
        }

        // Update motion parameters for both balls.
        a->checkMotion(time);
        b->checkMotion(time);
    }
```

5.4 Standard Types

Type representations vary between C++ implementations. For example, the type `int` may be represented as 16, 32, or 64 bits. Consequently, it is common practice for experienced C++ programmers to define a set of types for a program and use them throughout the program. Here is one such set (Reiss, page 50):

```
typedef  int           Integer;
typedef  char          Boolean;
typedef  char          Character;
typedef  double        Float;
typedef  char *        Text;
typedef  const char *  ConstText;
```

To see why this is helpful, consider the following scenario. Programmer A develops a program using these types for a system with 64-bit integers. The program requires integer values greater than 2^{32}. Programmer B is asked to modify the source code so that the program will run on processors for which `int` is represented by 32-bit integers. Programmer B changes one line

```
typedef long Integer;
```

and recompiles the program. Everything will work, provided that `long` is implemented using 64-bits. (If it isn't, B can type `long long` instead.)

Other applications include: a programmer using an up to date compiler can redefine `Boolean` as `bool` (which is included in the C++ Standard); and can redefine `Text` to allow unicode characters.

The need for `typedef` statements is perhaps less than it used to be, because there is less variation between architectures than there used to be. With the final demise of DOS and its segmented address space, most processors provide 32-bit integers and pointers. The distinction between 16 and 32 bits was important because $2^{16} = 65,536$ is not a large number. The

distinction between 32 and 64 bits is less important since $2^{32} = 4,294,967,296$ is large enough for many purposes. Since the introduction of `bool` (and the constants `true` and `false`), it is not really necessary to define `Boolean`. Nevertheless, you may wish to use `typedef`s to improve the portability of your code.

5.5 Pointers

In object oriented programming, we often need to construct composite objects. A *composite object* is an object that (conceptually) contains other objects. For example, a chess board "contains" squares and a solar system "contains" a sun, planets, satellites, and miscellaneous junk.

In C++, an object that contains other objects can actually contain them (that is, the contained objects are part of the containing objects memory space) or point to them. The difference is most easily seen from a classs declaration: a `Computer` actually contains (or *embeds*) a `Processor` but has a pointer to (or *references*) a `Memory`.

```
class Computer
{
    ....
    private:
        Processor proc;
        Memory *mem;
};
```

C++ is unusual amongst object oriented languages in providing this flexibility: Simula 67 is similar but more recent languages, such as Smalltalk and Java, provide pointers only. Eiffel provides both, but the default is for an object to contain pointers to other objects.

How do we choose whether to use embedding or pointers? It is best to use pointers unless there is a good reason not to do so. This choice has some important consequences:

- A pointer is not an object. The contained objects must be explicitly allocated on the heap using `new`.
- Since the contained objects are allocated on the heap, the must be explicitly destroyed using `delete`.
- In some situations, there may be several pointers to the same object. This is often desirable (sharing memory reduces memory requirements) but can also give rise to unexpected effects ("aliasing").

Reiss suggests using one name for instances of the class and another name for pointers to these instances. For example,

```
typedef KnightHeursticInfo * Knightheuristic;
```

This convention has advantages and disadvantages. In favour:

- the type of a pointer is a simple name, without `*` and so

- it is unlikely that we will forget to write the *.

Against:

- more names must be defined;
- the presence of the * reminds us that the variable is a pointer.

5.6 Constants

Constant definitions are an essential part of good programming. As a general rule, there should be no numeric literals (other than 0 and 1) anywhere in a program except as part of a constant definition. We do not write

```
char buffer[120];
ostrstream os(buffer, 120);
```

but, instead,

```
const int BUFFER_SIZE = 120;
....
char buffer[BUFFER_SIZE];
ostrstream os(buffer, BUFFER_SIZE);
```

where "...." indicates that the constant definition may be far away (perhaps in a header file) from its use.

Constant definitions are especially important for maintenance. A maintenance programmer must be able to assume that changing the definition above to

```
const int BUFFER_SIZE = 200;
```

will consistely change the size of all buffers in the program. This is a much safer change than using the editor to change all occurrences of "120" to "200", which might do considerable damage to the program.

It is also a good idea to define string literals as constants rather than including them in the code. Instead of

```
MessageBox(hwnd, "Your database was accidentally deleted", "Manager", MB_OK);
```

write

```
const char *APP_NAME = "Manager";
const char *DB_DEL = "Your database was accidentally deleted";
....
MessageBox(hwnd, DB_DEL, APP_NAME, MB_OK);
```

If you put all such text strings into a single file, you can easily put together versions of your program for different languages.

5.7 Class Declarations

The recommended form for a class declaration is as follows:

```
class Name
{
     public:

         Name ();  // constructor
         ~Name ();  // destructor

         // other public methods

    protected:

         // protected methods

         // protected attributes

    private:

         // private methods

         // private attributes

};
```

Blank lines should be included as shown. Each method (member function) or attribute (data member) should be accompanied by a comment.

An alternative approach is to place the private attributes first — Reiss considers that this "helps the reader to understand what the object is about". Private attributes, however, are not part of the interface and should not be emphasized.

It is best to include a constructor and a destructor even though C++ provides defaults. As we will see later, it is also a good idea to provide a copy constructor and an assignment operator, unless we do not want clients to make copies of instances of the class.

5.8 File Organization

The simplest and best rule for C++ programs is:

- one header file for each class declaration;
- one implementation file for the method definitions of each class;
- one header file (perhaps more for a large project) files containing information that is used throughout the program, such as commonly used constant definitions (but *not* global variables!); and
- one implementation file for the main program.

Header files have the suffix `.h` or `.hpp`. Header files may `#include` other header files but must *not* include implementation files. To prevent the compiler reading header information more than once, a header file should have the following structure:

```
#ifndef A_UNIQUE_IDENTIFIER
#define A_UNIQUE_IDENTIFIER

// Declarations

#endif
```

Implementation files have the suffix `.cpp`, `.C`, or `.cc`, depending on the system you are using. Implementation files are never included and do not need the directives shown above.

The following convention is simple and should be used wherever possible. For a class named `MyFirstClass`:

- Write a header file called `myfirstclass.h` with the following structure:

```
#ifndef MYFIRSTCLASS_H
#define MYFIRSTCLASS_H

class MyFirstClass
{
public:
    MyFirstClass ();
    ~ MyFirstClass ();

    // Declarations of other methods
    ....
};

#endif
```

- Write an implementation file called `myfirstclass.cpp` with the following structure:

```
#include "myfirstclass.h"

// #include directives for other application classes

// #include directives for standard header files

MyFirstClass::MyFirstClass ()
{
    ....
}

MyFirstClass:: ~ MyFirstClass ()
{
    ....
```

```
}

// Definitions of other methods of MyFirstClass
....
```

- Write an implementation file for the main program with an appropriate name for your application and the following structure:

```
#include "myfirstclass.h"

// #include directives for other application classes

// #include directives for standard header files

int main (int argc, char *argv[])
{
    MyFirstClass myObject(argc, argv);
    myObject.doIt();
    return 0;
}
```

Reiss says that you can put several class declarations in one file, and does so (for example, `knight_local.h` on pages 478–481). He also says (page 466):

> Overall, however, your `.cxx` and `.h` files should match up — this increases the symmetry and overall beauty of your project.

These statements seem inconsistent: how can you "match up" file names if the header files contains several class declarations but the implementation files contain definitions for only one class?

6 Using C++ Effectively

This section corresponds to Chapter 4 of Reiss's book.

C++ is a large and complex language. It is better to learn a subset of it well, and to use only that subset, than to attempt to learn the whole language. The notes in this section suggest conventions that will help you to make your C++ programs reliable, readable, and maintainable.

6.1 Objects or Pointers to Objects?

We have discussed this topic before (in Section 5.5) but return to it here in greater depth.

There are two patterns for using objects in C++. We can allocate them on the stack, like this:

```
(1)     {
(2)         ....
```

```
(3)          Course comp542;
(4)          ....
(5)      }
```

- Line (1) starts a new scope.
- Somewhere in this scope, at line (3), we declare an instance of class `Course`. At this point, the run-time system allocates space for an instance of class `Course` *on the stack* and calls the default constructor for the class. (If we had put arguments after `comp542`, the constructor with a matching parameter list would have been called.)
- Within the scope, we can use the object `comp542` (line (4)).
- At the end of the scope, line (5), the run-time system calls the destructor for class `Course`, deleting the object `comp542`.

Alternatively, we could have used the object `comp542` like this

```
(1)      {
(2)          ....
(3)          Course pcomp542 = new Course();
(4)          ....
(5)          delete pcomp542;
(6)          ....
(7)      }
```

- Line (1) starts a new scope, as before.
- At line (3), the run-time system allocates space for an instance of class `Course` *on the heap* and calls the default constructor for the class. The operator `new` indicates heap allocation.
- Later in the same scope, the `delete` statement calls the destructor for `pcomp542` and deallocates the memory it is using.

As we saw in Section 5.5, there is a similar choice to make when an object C is a client of a server object S. C may actually contain S or it may have only a pointer to S. In the first case, constructions and destruction of S are implicit and in the second case they are explicit (the programmer must write `new` and `delete` statement).

Here are some of the advantages and disadvantages of the two methods.

- Stack allocation means less work for the programmer. The constructor and destructor for the object are called automatically and all memory management is automatic.
- Stack allocation may lead to unnecessary copying. Here is a simple example.

```
(1)      Course copy (Course c)
(2)      {
(3)          return c;
(4)      }
```

```
(5)
(6)       Course comp542;
(7)       Course anotherCourse;
(8)       anotherCourse = copy(comp542);
```

Lines (6) and (7) call the constructor for class `Course` implicitly. In line (8), the argument `Comp542` is copied (using the copy constructor for class `Course`) into the stack frame for function `copy`. This value is then copied again into the result, `anotherCourse`. This code therefore contains four implicit calls to the constructor.

- We can avoid unnecessary copying if we pass arguments and return results by reference. If we change the code above to

```
(1)       Course & copy (Course & c)
(2)       {
(3)           return c;
(4)       }
(5)
(6)       Course comp542;
(7)       Course anotherCourse;
(8)       anotherCourse = copy(comp542);
```

  then line (8) will not cause any intermediate objects to be created by copying.

- Operators `new` and `delete` call functions that take time to execute. Heap allocation therefore takes longer than stack allocation, which is essentially free (the overhead is simply the adjustment of a pointer on entry to and exit from a function). However, this is not a serious objection to heap allocation because memory management is handled quite efficiently by most modern compilers.

- It is likely that there will be *some* heap allocation in all but the most trivial programs. For example, any program that uses a linked list, a tree, or any dynamic structure, must use the heap for nodes of the structure. It follows that, if we use stack allocation, most programs will have objects both on the stack *and* on the heap. This is confusing because the objects must be managed in different ways: implicit *versus* explicit allocation and deallocation, use of "`.`" *versus* "→", etc.

  If we allocate *all* objects on the heap, the program will be consistent: we will use explicit allocation and deallocation, "→", and so on, everywhere. Objects will not be copied unless we define and use copying methods.

- C++ uses virtual methods only when we provide a pointer or reference to an object. With stack allocation, we have to make sure that virtual methods are called when required; with heap allocation, virtual methods are called whenever possible.

- It is probably best to use stack allocation for very simple classes. Instances of the following classes, for example, would be allocated on the stack unless there was some good reason to allocate them on the heap:

```
class Point
{
```

```
    // An instance is a point in three dimensions with integer coordinates.
    public:
        ....
    private:
        int x;
        int y;
        int z;
};

class Complex
{
    // An instance is a complex number with double precision components.
    public:
        ....
    private:
        double real;
    double imag;
};

class Money
// An instance is an amount of money represented as integer cents.
{
    public:
        ....
    private:
        long amount;
};
```

[For classes like this, Reiss recommends using a `struct`. I prefer to use a class because there usually turn out to be some useful methods that we can associate with the data.]

6.2 Access Control

C++ provides fine control over access to the attributes and methods of a class. Any feature can be declared:

- `private`: only instances of the same class have access to the feature;
- `protected`: only instances of the same class and derived classes have access to the feature; and
- `public`: access to the class provides access to the feature.

Some languages provide even more security than C++. Smalltalk programmers are surprised that the following is acceptable in C++, because Smalltalk objects can only access their own attributes, not those of other instances of the same class. In Smalltalk, the expressions `other.real` and `other.imag` would have to be replaced by access functions.

```
Complex operator+ (Complex other)
```

```
{
    return Complex(real + other.real, imag + other.imag);
}
```

In C++, we can declare a class or feature to be a `friend` of another class. Friends have access to private components. This is a useful feature in special circumstances, but it should be used only when necessary. Most operators do not have to be declared as friends because they can be declared as member functions of the class. The stream insertion operator `<<` and extraction operator `>>` cannot be declared as member functions (because their first argument is a reference to a stream) and they are usually declared as friends.

In general:

- When declaring a feature, restrict its access as much as possible (that is, prefer `private` to `protected`, and prefer `protected` to `public`).
- Attributes (data members) should always be `private` or, if they are needed by derived classes, `protected`.
- Use the `friend` declaration only when there is no sensible alternative.

6.3 Using `const`

Consider the following declarations:

```
int size = 100;
const int CAPACITY = 500;
```

Clearly, we can use the `const` form only if we know that the program will never need to change the value of the variable. In this example, the program can change the value of `size` but not the value of `CAPACITY`. If we *do* know that a value will not change, there are several reasons for declaring it as `const`:

- `const` informs the reader that the value of `CAPACITY` never changes. (Of course, the convention that constants have upper-case names also helps the reader. But a convention is not the same as a language feature.)
- It is easy and safe to change the value of a constant, assuming that the programmer has used it consistently throughout the program.
- The compiler ensures that the value of constants does not in fact change: using `CAPACITY` in any context that would cause a new value to be assigned to it leads to a compile-time error message.
- The compile uses the `const`-ness of data to optimize. This applies especially to constant expressions. For example, if we write

```
const double PI = 4.0 * atan(1.0);
const double RADIUS = 12.0;
const double VOLUME = (4.0/3.0) * PI * RADIUS * RADIUS * RADIUS;
const double DENSITY = 3.57;
const double MASS = VOLUME * DENSITY;
```

then writing `MASS` in the program will generate exactly the same code as if we wrote `25840.5` (assuming that we are using a respectable compiler!).

Most programming languages provide declarations corresponding to `const` declarations in C++. However, C++ also provides a number of other uses of `const` that most other languages do not.

- Writing `const` after the parameter list in a method (member function) heading declares that the method does not change the state of the object. In the following example, `i++` is allowed because `i` is a local variable but `k++` is a compile-time error because `k` is an attribute.

```
class Example
{
    public:
        void fun () const
        {
            int i;
            i++;
            k++;
        }
    private:
        int k;
};
```

- Writing `const` before a parameter of a function declares that the function does not change the value of that parameter. In the following example, the call `fred.play()` is allowed only if the method `play()` has been declared `const` and the statement `num++` is not allowed.

```
class Example
{
    public:
        void fun (const Person fred, const int num)
        {
            fred.play();
            num++;
        }
};
```

- Writing `const` before the return type of a function declares that the result returned cannot be changed. In the following code, the assignment to `px` is not allowed because, by assigning to `*px`, we could change the value of the constant returned by `fun()`.

```
const double * fun ()
{
    return new double(5.0);
}
```

```
void main ()
{
    double *px;
    px = fun();
}
```

- Assuming the same definition of `fun()`, the assignment to `py` *is* allowed because it changes the *pointer* `py` but not the *value* `*py`.

```
void main ()
{
    const double *py;
    py = fun();
    double z;
    py = &z;
}
```

- However, if `const` occurs *after* the "`*`" in the declaration of `fun()`, then we are *not* allowed to change the pointer. In the following code, the assignment to `py` is *not* allowed.

```
const double * const fun ()
{
    return new double(5.0);
}

void main ()
{
    const double * const py = fun();
    double z;
    py = &z;
}
```

In general:

- Use `const` wherever possible in C++ programs.
- *Do not* assume that you can leave `const` declarations out during development and put them in later.

  Adding `const` declarations "later" is likely to be an extremely frustrating experience because each `const` that you add will probably create an inconsistency in the program and therefore one or more compiler-time errors.

6.4 Using `enum`

There are two common uses of `enum`: one is the "normal" use and the other avoids a defect in the design of C++. We discuss the second, and less important, use first.

C++ does not allow constant declarations within classes. You cannot write this:

```
class PrintManager
{
    ....
    private:
        const int BUFFERSIZE = 120;
        char buffer[BUFFERSIZE];
};
```

It is possible to get around this restriction by moving the constant declaration outside the class:

```
const int BUFFERSIZE = 120;

class PrintManager
{
    ....
    private:
        char buffer[BUFFERSIZE];
};
```

This is not a good solution, however, because it puts `BUFFERSIZE` at file scope. Since the class declaration is probably in a header file that is `#include`d by other files, `BUFFERSIZE` effectively becomes public.

Since C++ does allow `enum` declarations inside class declarations, we can solve the problem like this:

```
class PrintManager
{
    ....
    private:
        enum { BUFFERSIZE = 120 };
        char buffer[BUFFERSIZE];
};
```

The solution is rather clumsy, because `enum` was not intended for this purpose and the solution cannot be applied to types other than `int`.

The principal and more important use of enumerations is to declare a collection of integer constants. We do not care what values these constants have provided that they are all different. One common application of enumerations is to distinguish the states of a finite-state machine:

```
enum Colour { RED, GREEN, BLUE };
enum PrinterStatus { IDLE, PRINTING, NEEDS_PAPER, NEEDS_TONER, BROKEN };
enum ErrorType { OK, WARNING, SERIOUS, DISASTER };
```

The first of these is equivalent to

```
const int RED = 0;
const int GREEN = 1;
const int BLUE = 2;
```

and the others are similar. The name that follows `enum` behaves like a type, so that we can write:

```
Colour light;
PrinterStatus prState;
ErrorType kind;
```

Note that the syntax is the same as that of a class: the declarations

```
class OneType { .... };
enum AnotherType { .... };
```

introduce two types, `OneType` and `AnotherType`, into the program.

Unfortunately, C++ allows us to define the values of an enumeration explicitly. This is occasionally useful, as in the example of `BUFFERSIZE` above (although we used this only to compensate for another defect of C++!), and as in Reiss's example (page 98):

```
enum PinSide
{
    PIN_SIDE_NONE = 0;
    PIN_SIDE_FRONT = 0x1;
    PIN_SIDE_BACK = 0x2;
    PIN_SIDE_TOP = 0x4;
    PIN_SIDE_BOTTOM = 0x8;
    PIN_SIDE_LEFT = 0x10;
    ....
}
```

This example uses hexadecimal literals to emphasize the fact that the values are single bits in different positions of the word. This means that we can use bit-wise operators, as in `PIN_SIDE_TOP | PIN_SIDE_BOTTOM` (which has value `0xd`).

The problem with explicit enumerations is that they prevent us from treating the values of an enumeration as a sequence. For example, the following code is *not* allowed:

```
Printer laser2;
PrinterStatus state2;
for (state2 = IDLE; state2 <= BROKEN; state2++)
    laser2.testStatus(state2);
```

The reason is that, although in this case we know that the states have values 0, 1, 2, 3, and 4, the compiler cannot rely on this in general. If we changed the declaration of `PrinterStatus` to

```
enum PrinterStatus { IDLE, PRINTING, NEEDS_PAPER, NEEDS_TONER, BROKEN = 99};
```

the `for` loop would not work. Because it *might* not work, C++ compilers prevent us from using it at all. (Some compilers allow the loop but give a warning message. It is best not to use it anyway, however, because of the possibility of redeclarations such as `BROKEN = 99`.)

6.5 Parameters

C++ provides several ways of passing an argument to a function. We discuss the most common of them first. Assume that `Clock` is a class with a default constructor that has been declared in the current scope.

(a) Arguments can be *passed by value*. In the code below, C++ constructs a copy of `timePiece` on the stack. In the body of `adjust`, the name `clk` refers to this copy. Changes to `clk` do not affect `timePiece`.

```
void adjust (Clock clk)
{
    .... clk ....
}
....
Clock timePiece;
adjust(timePiece);
```

(b) If we want the function to change the value of its argument, we can pass a pointer to the argument rather than the argument itself. The standard name for this is *pass by reference* but this usage is confusing because pointers and references are different things in C++. Accordingly, we will call it *passing a pointer*. In the parameter list, we write "`*`" to indicate that the function expects a pointer. The caller must pass an address rather than an object. Within the function, `pClk` denotes the address of `timePiece` and `*pClk` denotes `timePiece` itself.

```
void adjust (Clock * pClk)
{
    .... *pClk ....
}
....
Clock timePiece;
adjust( & timePiece);
```

(c) C++ provides another way of passing arguments which corresponds more closely to *call by reference* in other languages. We write "`&`" after the type in the parameter list. The argument appears to be an object, although it is in fact an address. Within the function, `clk` denotes the clock itself, as in the first example.

```
void adjust (Clock & clk)
{
    .... clk ....
}
....
Clock timePiece;
adjust(timePiece);
```

Note that in (b), the "`&`" is the "address-of" operator of C but in (c) the same symbol is the reference operator of C++.

The second is historical accident: since C does not have references, we have to use pointers to simulate pass by reference in C. C++ provides references but is also supposed to be compatible with C; therefore it provides both pointers and references as ways of passing arguments.

If we follow the conventions outlined in Section 6.1, the most common way of passing arguments that we will use is probably a variant of the second one above: the argument is a pointer but, since we already have a pointer, we do not need the address-of operator:

```
void adjust (Clock * pClk)
{
    .... *pClk ....
}
....
Clock *pt = new Clock;
adjust(pt);
```

If we follow Reiss and use `typedef` definitions, our code will look more like this:

```
typedef  class ClockInfo *  Clock;
....
void adjust (Clock pClk)
{
    .... pClk->getTime(); ....
}
....
Clock pt = new ClockInfo;
adjust(pt);
```

Other argument passing mechanisms exist but are needed only rarely. One fairly common example is a function that adds an entry to a binary search tree. In the function `enter`, we must pass the argument corresponding to parameter `root` as a pointer to an `Entry` but, since the function may alter the value of the pointer, it must be passed by reference:

```
void enter (char *name, Entry * & root);
```

In summary:

- Pass basic types (`bool`, `char`, `double`) by value. (Exception: strings are often passed as `char *`.)
- Pass objects as references or pointers to avoid copying.
- Use `const` to indicate that a function does not change its parameters.
- Avoid complicated parameter declarations when possible.

6.6 Overloading

Overloading is a controversial but very useful feature of C++. Used carefully, overloading contributes to the clarity of a program; used haphazardly, it can create tremendous confusion.

In earlier languages, function calls were very simple. When you see an expression like `f(x)`, you look for the declaration of `f` and you know immediately what is happening. In C++, you are more likely to encounter something like `o→f(x)` and it may be far from obvious which version of `f()` is being invoked:

- There may be several declarations of `f()` with different parameter lists.
- The parameter lists may contain defaults. That is, `f(x)` may actually be calling a function declared as

  ```
  void f(int x, int y = 0);
  ```

  but the second argument is defaulted.
- Suppose that there does not appear to be a match between `f(x)` and *any* of the declarations of `f()`. Then it is possible that C++ has implicitly converted `x` to some other type and called a function `f()` defined for that type. This is particularly likely to happen with constructors. Suppose that a class `Table` has a constructor

  ```
  Table::Table (int size);
  ```

  that constructs an empty `Table` with space for `size` entries. Suppose also that there is a function

  ```
  void print (Table tbl)
  ```

  that prints a `Table`. The statement `print(5)` will create an empty table with five entries and print it!
- The function may have been declared `virtual` in a base class. In this case, the version invoked depends on the class of `*o`.

These examples illustrate the problems that can arise from overloading. To see the advantages, consider the following calls to the OpenGL library:

```
glVertex2f(1.0, 2.0);
glVertex3f(1.0, 2.0, 3.0);
glVertex4i(1, 2, 3, 4);
glVertex3fv(pVec);
```

These are four of the 24 functions that define a vertex. Each name corresponds to a different way of providing the arguments: two floats, three floats, four integers, address of vector of three floats, etc. In C++, there would be one function called `glVertex()` and it would be overloaded for different kinds of arguments.

C++ allows operators to be overloaded. Overloaded operators should be used for:

- "mathematical" classes, such as `Complex`, `Vector`, `Matrix`, and `Polynomial`, for which there are natural meanings of `+`, `-`, etc;
- classes for which stream input and output is best provided by overloading the insertion operator `<<` and the extraction operator `>>`;

- iterator classes defined in such a way that loops have the standard form

```
for (Iterator iter(collection); !iter; iter++)
    ....
```

  (Note that Reiss would write the termination condition as `iter` rather than `!iter`. This works by overloading the conversion function `int()` and seems to me to be confusing rather than helpful.)

In general: use overloading, but use it carefully to improve the clarity and readability of your programs.

7 Designing a Class

Every class is different and has its own special requirements. Nevertheless, it is a good idea to have an idea of what a typical class should look like. Then we can vary the general pattern when there is good reason to do so.

7.1 Constructors

Here are some of the factors to bear in mind when you are designing constructors for a class:

- If a class does not have a constructor, the compiler will generate a default constructor without parameters. This constructor doesn't do anything useful; in particular, it doesn't initialize the attributes of the object.
- There are a few situations in which C++ will generate calls to the default constructor. For example, when the run-time system processes a declaration of an array of objects, it calls the default constructor for each object.
- You may want to re-initialize an object that already exists so that you can use it again. You cannot call the constructor of an existing object, but you can call an initialization function. The constructor can call the same function.
- Constructors can have default parameters; this feature can be used — with discretion — to reduce the number of constructors without reducing flexibility.
- C++ uses constructors as conversion functions, as discussed in Section 6.6 above.

These considerations suggest the following. For each class:

- Write an initialization method.
- Write a default constructor that calls the initialization method.
- Write as many other constructors as necessary.

The following code for class `Person` illustrates these ideas. The second constructor makes its own copies of the strings passed to it. This ensures that an instance of `Person` does not contain a pointer to a local buffer that may get overwritten or destroyed. (Passing instances of the standard `string` class by value would have the same effect.)

```
class Person
{
    public:
        Person ();
        Person (char *name = NULL, char *email = NULL, int age = 0);
        void initialize ();
    private:
        char *d_name;
        char *d_email;
        int d_age;
};

Person::Person ()
{
    initialize();
}

Person::Person (char *name, char *email, int age)
{
    if (name != NULL)
    {
        d_name = new char[strlen(name) + 1];
        strcpy(d_name, name);
    }
    if (email != NULL)
    {
        d_email = new char[strlen(email) + 1];
        strcpy(d_email, email);
    }
    d_age = age;
}

void Person::initialize ()
{
    d_name = NULL;
    d_email = NULL;
    page = 0;
}
```

7.2 Destructors

If you don't declare a destructor, the compiler will generate one that does nothing. This is acceptable only if the object doesn't have any pointers. The following code uses the class `Person` defined above and has a memory leak because, at the end of the scope, two strings are not deallocated:

```
void main ()
{
```

```
    Person("Pixie Penguin", "pix@antarctica.org", 8);
    ....
}
```

The solution, for this class, is to define a destructor:

```
Person:: ~ Person ()
{
    delete [] d_name;
    delete [] d_email;
}
```

In general:

- Define a destructor for every class. Even if the class doesn't have pointers now, they may be added later.
- The destructor should delete any objects that this object has pointers to *provided that it owns them.*
- When several objects "share" another object, in the sense that they possess pointers to it, one of them should be given the responsibility of deleting it.
- A `delete` statement must contain "[ ]" if and only if the corresponding `new` statement contains "[ ]".

7.3 Copy Constructors

A *copy constructor* is a constructor that makes a new instance of a class by copying an existing instance. A copy constructor has one parameter, which is a reference to the class. The copy constructor for class `Person` is declared like this:

```
Person (Person & other);
```

If you do not declare a copy constructor, the compiler will generate one for you. If your class contains pointers, the generated copy constructor is a source of trouble. Suppose that we have not defined a copy constructor for class `Person` and the program contains this code:

```
void sendMessage (Person p)
{
    ....
}

void someFunction ()
{
    Person jean;
    sendMessage(jean);
}
```

C++ uses the default copy constructor to initialize the local object `p`. It does this by copying the pointers, not the strings pointed to. Consequently, `p` contains pointers to the same strings (`name` and `email`) as `jean`. At the end of the scope of `sendMessage`, `jean` is deleted — taking these strings with it!

It is a good idea to define a copy constructor for any class that has pointers. Here is a copy constructor for class `Person`:

```
Person (Person & other)
{
    d_name = new char[strlen(other.d_name) + 1];
    strcpy(d_name,other.d_name);
    d_email = new char[strlen(other.d_email) + 1];
    strcpy(d_email,other.d_email);
    d_age = other.d_age;
}
```

7.4 Assignment Operators

Assignment operators have the same problem as copy constructors: the compiler generates one automatically, and it is usually wrong. The following code has several errors:

```
void main ()
{
    Person bill ("William Shakespeare", "bill@globe.com", 435);
    Person will;
    will = bill;
}
```

The first declaration allocates space for two strings. The assignment statement makes copies of pointers to these strings but not of the strings themselves. At the end of the scope, both `bill` and `will` are deleted, and the two strings are deleted *twice*.

Coding assignment operators requires care. We will develop an assignment operator for class `Person` in stages. Here is the first version:

```
Person & Person::operator= (const Person & rhs)
{
    d_name = new char[strlen(rhs.d_name) + 1];
    strcpy(d_name,rhs.d_name);
    d_email = new char[strlen(rhs.d_email) + 1];
    strcpy(d_email,rhs.d_email);
    d_age = rhs.d_age;
    return *this;
}
```

The reason that the assignment operator returns a value is that C++ (following C) allows assignment sequences such as `p = q = r`. Since this is parsed as `p = (q = r)`, the assignment `q = r` must return a result that becomes the value of `p`.

The obvious problem with the version above is that the strings belonging to the current object are not deleted. This is corrected in the second version:

```
Person & Person::operator= (const Person & rhs)
{
    delete [] d_name;
    d_name = new char[strlen(rhs.d_name) + 1];
    strcpy(d_name,rhs.d_name);
    delete [] d_email;
    d_email = new char[strlen(rhs.d_email) + 1];
    strcpy(d_email,rhs.d_email);
    d_age = rhs.d_age;
    return *this;
}
```

This leaves a more subtle problem: consider the effect of the assignment `p = p` in which `p` is an instance of `Person`. The first thing that happens is that the `name` and `email` strings of `p` are deleted! (An assignment of the form `p = p` is unlikely but, since it is legal, it should work correctly. Such assignments can occur in disguised form, such as `people[i] = people[j]`, where `i` and `j` happen to have the same value.)

To make this assignment work, we have to check for this special case:

```
Person & Person::operator= (const Person & rhs)
{
    if (this == &rhs)
        return *this;
    delete [] d_name;
    d_name = new char[strlen(rhs.d_name) + 1];
    strcpy(d_name,rhs.d_name);
    delete [] d_email;
    d_email = new char[strlen(rhs.d_email) + 1];
    strcpy(d_email,rhs.d_email);
    d_age = rhs.d_age;
    return *this;
}
```

7.5 Avoiding Trouble

The previous sections demonstrate that the default methods created by the compiler are dangerously wrong in the presence of pointers. It is tedious to write all of the functions described above simply to avoid problems created by the compiler.

We can avoid having to write these functions by noting that the problems arise *only when objects are copied*. That is, when we construct new objects using the copy constructors, pass objects to functions by value, and assign objects. All of the problems can be avoided if we work with *pointers to objects* rather than the objects themselves. In other words:

- Allocate new objects using `new` and deallocate them using `delete`.
- Assign pointers, not objects.
- Pass objects as pointers or by reference.

Unfortunately, these conventions make memory management more complicated, because we must ensure that each object is deallocated exactly once. On the other hand, we gain efficiency because we do not have to copy objects.

If we want to be really safe, we can *prevent* objects being copied or assigned. The way to do this is to define the operations but make them `private`. Here is an appropriate declaration of class `Person` that does this. The assignment operator returns `*this` to avoid error messages from the compiler.

```
class Person
{
    public:
        Person ();
        Person (char *name = NULL, char *email = NULL, int age = 0);
        void initialize ();
    private:
        Person (Person &) {}
        Person & operator= (const Person & rhs) { return *this; }
        char *d_name;
        char *d_email;
        int d_age;
};
```

8 Review of Inheritance

In this section, we review the basic ideas of inheritance. The material in this section is based on a (somewhat artificial) running example.

We assume that there is a small airline that organizes its aircraft using an inheritance hierarchy. The classes are:

```
Aircraft
Jet
Jumbo
Stretch
```

in which `Aircraft` is the base class, `Jet` is derived from `Aircraft`, and so on.

Here is the declaration of the base class `Aircraft`. Note that the attribute `lengthAircraft` is `protected` rather than `private` because derived classes must access it.

```
class Aircraft
{
    public:
        Aircraft ();
        int length ();
        int profit (double dist);
        int revenue (double dist);
        int expense (double dist);
```

```
        double basicRate ();
        virtual int capacity (double dist) = 0;
        virtual int price (double dist) = 0;
        virtual int consumption (int numberPassengers) = 0;
    protected:
        int lengthAircraft;
};
```

Here are the function definitions for class `Aircraft`. The constructor in each derived class initializes the data member `lengthAircraft`. Without inheritance, the data member `lengthAircraft` and the function `length()` would have to be defined in every class.

```
Aircraft::Aircraft ()
{
    lengthAircraft = 100;
}

int Aircraft::length ()
{
    return lengthAircraft;
}
```

In the next few methods, note that we are calling methods that are not implemented in class `Aircraft`, although they have been declared as pure virtual.

```
int Aircraft::profit (double dist)
{
    cout << "Aircraft  profit" << endl;
    return revenue(dist) - expense(dist);
}

int Aircraft::revenue (double dist)
{
    cout << "Aircraft  revenue" << endl;
    return capacity(dist) * price(dist);
}

int Aircraft::expense (double dist)
{
    cout << "Aircraft  expense" << endl;
    return dist * consumption(capacity(dist));
}

double Aircraft::basicRate ()
{
    cout << "Aircraft  basicRate" << endl;
    return 0.1; // dollars per kilometre
}
```

`Aircraft` is an *abstract class*: the compiler will not allow instances to be constructed. Nevertheless, `Aircraft` *does* have a constructor. The constructor `Aircraft::Aircraft` is called whenever an instance of a derived class in being constructed.

8.1 Inheritance avoids duplication

The first function that we consider is `length()`. This function is declared and defined in the base class `Aircraft` and used in all derived classes.

Suppose we declare

```
Jet a;
Jumbo b;
```

Then the expressions

```
a.length();
b.length();
```

perform the same computation, but may give different results, depending on the value of `lengthAircraft` in the class of the receiver. Of course, this is nothing special, because attributes can vary from object to object, not just from class to class.

8.2 Functions can be redefined in derived classes

The `capacity` of an aircraft is the number of passengers that it can carry. We assume that the capacity depends on the distance that the aircraft has to travel, and that the formula used depends on the kind of aircraft. In class `Aircraft`, the function `capacity()` is declared as a **pure virtual** function: the keyword `virtual` makes it virtual, and the final expression "`= 0`" makes it "pure".

A **virtual** function is a function that can be redefined in derived classes and, through dynamic binding, called through a pointer to the base class. A **pure** virtual function is a virtual function that cannot be called in the class in which it is defined. (The notation `= 0` suggests a `NULL` pointer, indicating that the function has no implementation.)

A class that contains one or more pure virtual functions is called **abstract**. An abstract class cannot have instances. (If it had instances, it would be possible to invoke a pure virtual function, leading to disaster.)

The function `capacity()` is inherited and redefined in the derived classes. The formula for computing the capacity of a `Jet` is

$$N = 350\left(1 - \frac{D}{7500}\right)$$

and the formula for computing the capacity of a `Jumbo` is

$$N = 500\left(1 - \frac{D}{15000}\right)$$

In each case, N is the number of passengers that can be carried for a distance D kilometres.

The function `consumption()` computes the number of litres of fuel that an aircraft requires to fly one kilometre. Like `capacity()`, the calculation depends on the kind of aircraft. For a `Jet`,

$$C = 100 + N/4$$

and, for a `Jumbo`,

$$C = 150 + N/5$$

where C is the consumption and N is the number of passengers. (These values are rather unrealistic, but that does not affect the points illustrated by this example.)

Class declaration for `Jet` (note redefinition of pure virtual functions):

```
class Jet : public Aircraft
{
    public:
        int capacity (double dist);
        int price (double dist);
        int consumption (int numberPassengers);
};
```

Function definitions for class `Jet`:

```
int Jet::capacity (double dist)
{
    cout << "Jet       capacity" << endl;
    return int(350.0 * (1.0 - dist / 7500.0));
}

int Jet::price (double dist)
{
    cout << "Jet       price" << endl;
    return int(basicRate() * dist);
}

int Jet::consumption (int numberPassengers)
{
    cout << "Jet       consumption" << endl;
    return 100 + numberPassengers / 4;
}
```

Class declaration for `Jumbo`:

```
class Jumbo : public Jet
{
    public:
        Jumbo ();
        int capacity (double dist);
        int consumption (int numberPassengers);
};
```

Function definitions for class `Jumbo`:

```
Jumbo::Jumbo ()
{
    lengthAircraft = 120;
}

int Jumbo::capacity (double dist)
{
    cout << "Jumbo     capacity" << endl;
    return int(500.0 * (1.0 - dist / 15000.0));
}

int Jumbo::consumption (int numberPassengers)
{
    cout << "Jumbo     consumption" << endl;
    return 150 + numberPassengers / 5;
}
```

Using the declarations of Section 8.1, the expressions

```
a.capacity();
b.capacity();
```

perform different computations, depending on the definition of the function `capacity()` in the class of the receiver, and will probably give different results. Note that this is *not*dynamic binding; it is simply a consequence of the fact that `a` and `b` belong to different classes.

8.3 Base class functions can call derived functions

The function `expense()` is declared and defined in class `Aircraft`. Its definition uses the functions `capacity()` and `consumption()` that are declared as pure virtual functions in class `Aircraft` and are only given definitions in derived classes.

The evaluation of the expression

```
b.expense(2000.0)
```

assuming `b` is an instance of `Jumbo` calls the following functions:

```
Aircraft::expense()
    Jumbo::capacity()
    Jumbo::consumption()
```

The virtual function mechanism allows us to write code in a base class that calls functions in derived classes — functions that may not even have been written at the time the base class was defined.

We could introduce another class into the hierarchy, with definitions for `capacity()` and `consumption()`. The function `expense()` would work correctly for the new class. We can

achieve this by compiling the new class and relinking: the class `Aircraft` does not even have to be recompiled.

The function `revenue()` is similar to `expense()`: it is defined in class `Aircraft` but uses the pure virtual functions `capacity()` and `price()`.

The function `profit()` in class `Aircraft` computes the profitability of a flight by computing the excess of revenue over expense for the flight.

Consider the evaluation of the expression `a.profit(2000.0)` where `a` is an instance of `Jet`. These functions are called:

```
Aircraft::profit()
    Aircraft::revenue()
        Jet::capacity()
        Jet::price()
    Aircraft::expense()
        Jet::capacity()
        Jet::consumption()
```

Declaration of class `Stretch`:

```
class Stretch : public Jumbo
{
    public:
        Stretch ();
        int price (double dist);
};
```

Function definitions for class `Stretch`:

```
Stretch::Stretch ()
{
    lengthAircraft = 140;
}

int Stretch::price (double dist)
{
    cout << "Stretch   price" << endl;
    return int(2.0 * basicRate() * dist);
}
```

The functions called by the expression `s.profit(2000.0)`, where `s` is an instance of `Stretch`, are shown below. The calling pattern is the same, but the functions actually called are different.

```
Aircraft::profit()
    Aircraft::revenue()
        Jumbo::capacity()
        Stretch::price()
    Aircraft::expense()
        Jumbo::capacity()
        Jumbo::consumption()
```

8.4 Derived class functions can call base class functions

The account of `profit()` in Section 8.3 was slightly simplified. The function `price()` is a pure virtual function in the base class `Aircraft`; accordingly, it is redefined in derived classes. But the redefined functions call the function `basicRate()` in class `Aircraft`! The idea is that there is a standard base rate (10 cents per kilometre) and that this rate is scaled to obtain the price in the other classes where `price()` is redefined.

Class `Jet` computes the price by multiplying the distance by the basic rate; class `Jumbo` performs the same computation; and class `Stretch` multiplies the distance by twice the basic rate. The functions called by `s.profit()`, where `s` is an instance of `Stretch` are:

```
Aircraft::profit()
    Aircraft::revenue()
        Jumbo::capacity()
        Stretch::price()
            Aircraft::basicRate()
    Aircraft::expense()
        Jumbo::capacity()
        Jumbo::consumption()
```

We see that calls can traverse the hierarchy in both directions. In Section 8.3, base class functions called dervied class functions. In this section, derived class functions call base class functions.

This is a rather simple example in a relatively small hierarchy. In a large and complex program, such calls can be difficult to understand and trace. Because of these difficulties, this calling pattern has been called the "yoyo problem".

8.5 Function binding can occur at run-time

Here is the main program for `Aircraft`:

```
void main ()
{
    const int FLEET_SIZE = 3;
    Aircraft *fleet[FLEET_SIZE];
    fleet[0] = new Jet;
    fleet[1] = new Jumbo;
    fleet[2] = new Stretch;

    while (true)
    {
        cout << "Enter distance (<= 0 terminates): ";
        double distance;
        cin >> distance;
        if (distance <= 0.0)
            break;
        for (int i = 0; i < FLEET_SIZE; i++)
        {
```

```
            int capacity = fleet[i]->capacity(distance);
            double profit = fleet[i]->profit(distance);
            cout <<
                "Plane " << i <<
                " carries " << capacity <<
                " passengers with a ";
            if (profit >= 0.0)
                cout << "profit of $" << profit;
            else
                cout << "loss of $" << -profit;
            cout << '.' << endl;
        }
    }
}
```

The `for` loop in the main program computes and reports the capacity and the profit for each member of a fleet of aircraft. The fleet is stored in an array of pointers to `Aircraft`, but the objects pointed to are instances of `Jet`, `Jumbo`, and `Stretch`. The key expression,

```
fleet[i]->profit(D)
```

performs a computation in which the functions called depend upon the type of the object that `fleet[i]` points to. Since the compiler cannot determine (in the general case) what these types are, the name of the function must be matched to its definition while the program is running. This is called **dynamic binding**.

The program also reports the functions used in each computation. Here is a report generated by the program. Note that the range of 7500 kilometres is too long for a Jet and unprofitable for a Jumbo. The results were obtained by running the version of the program provided on the web page, which is slightly different from the version given in these notes.

```
Enter distance (<= 0 terminates): 7500
Jet       capacity
Aircraft  profit
Aircraft  revenue
Jet       capacity
Jet       price
Aircraft  basePrice
Aircraft  expense
Jet       capacity
Jet       consumption
Plane 0 carries 0 passengers with a loss of $375000.
Jumbo     capacity
Aircraft  profit
Aircraft  revenue
Jumbo     capacity
Jet       price
Aircraft  basePrice
Aircraft  expense
Jumbo     capacity
```

```
Jumbo     consumption
Plane 1 carries 249 passengers with a loss of $183750.
Jumbo     capacity
Aircraft  profit
Aircraft  revenue
Jumbo     capacity
Stretch   price
Aircraft  basePrice
Aircraft  expense
Jumbo     capacity
Jumbo     consumption
Plane 2 carries 249 passengers with a profit of $283125.
```

8.6 Conclusions

Inheritance, like any complex mechanism, has both advantages and disadvantages. Advantages include the following.

- Inheritance reduces code duplication by allowing common code to be moved to base classes.
- We can put code in the class where it belongs; general functions appear near the root of the class hierarchy and more specialized functions appear near the leaves.
- We can build programs incrementally.

  Instead of changing an existing class, we can derive a new class from it. Since the base class is not changed, there is no danger of breaking its existing clients. Sometimes, it is not even necessary to recompile the base class; we can just compile the derived class and relink the application.

  This is an important form of **code reuse** — we can use code in the base class more than once.

Inheritance also has some potential disadvantages.

- Dynamic binding slows the program down. Although this is a reason that is often given for not using object oriented techniques, it is actually a myth: the overhead of dynamic binding in languages like C++ and Eiffel is negligible.

  It is true that Smalltalk performance suffers because of dynamic binding. But Smalltalk is an untyped language in which the run-time system must perform a search up the class hierarchy at run-time. In typed, compiled languages, the search is done at compile-time and the overhead is usually no more than one indirection through a pointer.
- Calling constructors and destructors slows the program down. Unfortunately, this is true, especially for C++. The fault is not entirely due to inheritance, but calling a constructor (or destructor) in a derived class `C` can potentially call constructors (or destructors) in every class above `C` in the class hierarchy.

- Code can be hard to understand and trace. This is a serious problem; there are numerous anecdotes about hard-to-find bugs in object oriented code. Part of the problem is that there are not many debuggers that have been specifically tailored to handle the particular problems of debugging object oriented code. The main problem is dynamic binding. Suppose that you are trying to debug this function:

```
void func (Aircraft *p)
{
    .....
    p->profit();
    .....
}
```

  Notice that there is no way you can find out what `profit()` actually does without examining **calls** to `func` to find out the type of arguments used by its clients. In object oriented programming, a function is not a self-contained entity that can be understood in isolation from its environment.

9 A Natural Hierarchy

The best applications of inheritance occur when we have a *natural hierarchy* of classes. The members of a natural hierarchy form a tree with the most general class as its root. For example, `Vehicle` might be the root of a hierarchy containing classes such as `Car`, `Van`, `Bus`, and `Truck`. As another example, `Transaction` might be the root of a hierarchy containing classes such as `ChequePayment`, `CashPayment`, `Deposit`, `Adjustment`, and so on.

Classes is a hierarchy are related by *"is a kind of"*. (Note that "is a" can be misleading because we say, for instance, "Montreal is a city". In general, "A is a B" may mean "A is an instance of class B". But when we say "A is a kind of B" it is more likely that A and B are both classes.)

In the first example above, it makes sense to say "a car is a kind of vehicle", etc.

The natural hierarchy that we will discuss in this section has `Expression` as its root and various kinds of expression — `Constant`, `Variable`, `Operator`, and so on — as other members of the hierarchy.

The problem that we consider, based on Reiss (Chapter 5) is *Symbolic Differentiation*.

9.1 Symbolic Differentiation

Differentiation is an operation applied to functions. The result of differentiating the function $f(x) = ax^2+bx+c$ is another function $f'(x) = 2ax+b$. "Symbolic" simply means that we will be working with expressions like this rather than with the numerical values of the functions. We adopt some conventions:

- Differentiation is "with respect to" a particular variable. We will always differentiate with respect to x.

- We will use D to denote the operator $\frac{\partial}{\partial x}$. We could have written the example above as $D(ax^2 + bx + c) = 2ax + b$.

The reason for choosing differentiation as a problem rather than, say, integration, is that differentiation is defined by a simple set of recursive rules. It is therefore a completely mechanical operation. (Actually, integration has also been a completely mechanical operation since the introduction of Risch's algorithm, but the mechanics are much more complicated.) Although the algorithm for differentiation is simple, the details can become quite complex. Consider, for example:

$$\begin{aligned} D\left(\frac{\sin(k*x)*e^{-a*x^2*\ln(x)}}{a^2+x^2}\right) = & \quad \frac{\cos(k\,x)\,k\,e^{(-a\,x^2\,\ln(x))}}{a^2+x^2} \\ + & \quad \frac{\sin(k\,x)\,(-2\,a\,x\,\ln(x) - a\,x)\,e^{(-a\,x^2\,\ln(x))}}{a^2+x^2} \\ - & \quad 2\,\frac{\sin(k\,x)\,e^{(-a\,x^2\,\ln(x))}\,x}{(a^2+x^2)^2} \end{aligned}$$

We will use the rules below for differentiation. k stands for any constant; y stands for any variable other than x; u and v stand for expressions that may contain x. In the last rule, n is an integer that does not depend on x and is not -1.

$$\begin{aligned} D(k) &= 0 \\ D(y) &= 0 \\ D(x) &= 1 \\ D(u+v) &= D(u) + D(v) \\ D(u-v) &= D(u) - D(v) \\ D(uv) &= v\,D(u) + u\,D(v) \\ D(u/v) &= \frac{v\,D(u) - u\,D(v)}{v^2} \\ D(u^n) &= nu^{n-1}D(u) \end{aligned}$$

Here is an example of the rules being applied to a simple expression:

$$\begin{aligned} D\left(\frac{a}{x^2}\right) &= \frac{x^2D(a) - aD(xx)}{x^4} \\ &= \frac{0.x^2 - a(xD(x) + xD(x))}{x^4} \\ &= \frac{0.x^2 - a(x.1 + x.1)}{x^4} \\ &= \frac{a}{x^3} \end{aligned}$$

Note that simplification is required to obtain the last line: without simplification, our program would output the line before. Examples of simplification rules include:

$$\begin{aligned} u + 0 &\rightarrow u \\ 0 + u &\rightarrow u \\ u - 0 &\rightarrow u \end{aligned}$$

$$\begin{aligned} u \times 1 &\rightarrow u \\ 1 \times u &\rightarrow u \\ u/1 &\rightarrow u \\ u - u &\rightarrow 0 \\ u/u &\rightarrow 1 \end{aligned}$$

We could use character strings to represent expressions. For example, we could represent $ax^2 + bx + c$ as the string `"a*x^2+b*x+c"`, introducing explicit symbols for multiplication and powering. However, such strings are very awkward to work with, and it is easier to use a data structure. The natural data structure is a tree: variables and constants are leaves, unary operators are internal nodes with one subtree, and binary operators are internal nodes with two subtrees.

The program will work as follows:

1. read an expression from the user;
2. convert the expression into a tree;
3. differentiate the expression;
4. simplify the result of differentiating;
5. convert the tree back into a string;
6. display the string.

An expression will be represented as an object belonging to a particular class. The class will be chosen from a hierarchy (Reiss, page 109):

```
Expression
    Operator
        Unary Operator
            Minus
        Binary Operator
            Add
            Subtract
            Multiply
            Divide
            Power
    Leaf
        Variable
        Constant
```

The base class will define the major operations on trees as pure virtual functions, to be redefined appropriately in the derived classes. We will decide on a suitable return type for `differentiate()` and `simplify()` later.

```
class Expression
{
```

```
    public:
        type differentiate() = 0;
        type simplify() = 0;
        void unparse() = 0;
};
```

Note that we have *not* included `parse` as a method. This is because parsing is something best done from outside the class rather than inside it. We cannot do anything with a class until we have constructed an instance of it. If a class contains a parser, we have to know what kind of syntactic object we are reading *before* we can parse it. This is not always possible, as we can see from an expression as simple as $x + a$. It looks as if we are parsing a variable, x, but in fact we are parsing a binary operation, $x + a$.

Converting a string into a tree is called *parsing* and turning a tree back into a string is called *unparsing* or *formatting*. The program has to implement the following operations:

- parsing;
- differentiating;
- simplifying;
- unparsing.

We must make an important decision: do we alter trees "in place" or do we construct new trees?

- Altering trees in place saves storage and is feasible for simple expressions. For example, we can replace x by 1 and 56 by 0.
- Altering trees in place is going to lead to severe problems with more complicated expressions. For example, $D(ax) = 0.x + a.1$: a multiply node is replaced by an add node whose subtrees are multiply nodes.

We could attempt to minimize storage by sharing parts of trees. For example, the tree produced by differentiating ax could have pointers to a and x in the original tree (although x will eventually be removed during simplification). Sharing, however, is going to introduce complications in memory management. (This problem would not arise in languages with garbage collection, such as Java, Eiffel, and Smalltalk).

Based on these considerations, and the fact that we are writing the program in C++, we will use the following rule: differentiation and simplification accept a tree as input and return a *completely new tree* as output.

This means that we will have to implement a `copy()` function for all node types.

Should we delay simplification until we have finished differentiating, or should we simplify as we proceed? We have seen that differentiation without simplification leads to complex expressions and it would be a waste of time to differentiate unsimplified expressions. Further thought shows that this is not in fact a problem.

- Differentiation proceeds from the leaves to the root of the tree, so in fact we never differentiate anything twice and, in particular, we never differentiate an unsimplified expression.

- During differentiation, we will be constructing trees. We can incorporate some simplification into the constructors.

To see how these ideas work in practice, we will consider some of the methods of class `Multiply`. We assume that instances of this class have two attributes, `left` and `right`, pointing to expressions. Recall that $D(uv) = vD(u) + uD(v)$. Note that we do *not* use constructors directly.

```
Expression *Multiply::differentiate ()
{
    return makeAdd
        (
            makeMultiply(right->copy(), left->differentiate()),
            makeMultiply(left->copy, right->differentiate())
        );
}
```

Next, consider how the method `makeMultiply` will work. We can assume that the arguments passed to it are fresh trees that are not part of any existing expression.

```
Expression *makeMultiply (Expression *lhs, Expression *rhs)
{
    if (isZero(lhs) || isZero(rhs))
    {
        delete lhs;
        delete rhs;
        return new Number(0);
    }
    if (isOne(lhs))
    {
        delete lhs;
        return rhs;
    }
    if (isOne(rhs))
    {
        delete rhs;
        return lhs;
    }
    if (isNum(lhs) && isNum(rhs))
    {
        double product = lhs->getValue() * rhs->getValue();
        delete lhs;
        delete rhs;
        return new Number(product);
    }
    return new Multiply(lhs, rhs);
}
```

Methods like `isNum()`, `isZero()` and `isOne()` must be defined in the base class `Expression()`. If we defined it as pure virtual, we would have to redefine it in every class. It makes more

sense to define it as a function returning `false` and to redefine it in class `Number`, because this is the only place where it can be true.

```
virtual bool Expression::isNum ()
{
    return false;
}

virtual bool Expression::isZero ()
{
    return false;
}

bool Number::isNum ()
{
    return true;
}

bool Number::isZero ()
{
    return value == 0.0;
}
```

Using these techniques, we can write a program that simplifies as it proceeds. In fact, whenever it constructs a new expression, it attempts to make it as simple as possible. There are still some hard decisions to make. Consider the "maker" for `Subtract`:

```
Expression *makeSubtract (Expression *lhs, Expression *rhs)
{
    if (lhs->equals(rhs))
    {
        delete lhs;
        delete rhs;
        return new Number(0);
    }
    .....
}
```

How difficult is it going to be to construct the function `equals`? If we do implement it, what sort of testing should it do?

9.2 Simplification: Reiss's Design

Reiss proposes an alternative, and more flexible, approach to simplification than the one described above. The idea is to provide a set of *simplification rules* rather than build the simplification strategies into the program. Reiss does not elaborate on this design (aprt from including the classes `RuleSet` and `Rule` in the design), but the intention is fairly clear.

- Rules could be provided in a text file. The first ffew rules might look like this:

```
A + 0 -> A;
A - 0 -> A;
A * 1 -> A;
....
```

Within a rule, the letter `A` stands for an arbitrary expression.

- The program can read the rule file and convert the rules into trees. The trees are similar to the expression trees used by the differentiator, but they have an additional operator corresponding to `->` in the rules. This operator is always at the root of the tree. In other words, a rule tree has a *left side* (e.g., `A + 0`) and a *right side* (e.g., `A`).
- Given an expression E in the form of a tree, the simplifier works as follows:
  - Attempt to match each node in E to the left side of a rule. For example, the node `x * y + 0` matches `A + 0`, the left side of the first rule above. (Note, however, that *trees* are matched, not text strings.)
  - The matching process binds the variable `A` in the rule to a subtree of the expression. For the example given, `A` is bound to the subtree `x * y`.
  - The output of the simplifier is the right side of the rule with occurrences of `A`, if any, replaced by whatever `A` is bound to. In the example, the output would be `x * y`.
- The simplifier builds a new tree to avoid memory management problems. For each node of the input expression, it returns either a deep copy of the subtree rooted at that node or, if it found a matching rule, a newly created subtree corresponding to the right part of the rule.

We will not discuss the differentiation program further. In the next section, we focus on general factors related to inheritance.

9.3 Designing with Inheritance

Use inheritance when you have a collection of classes that are naturally related by *is a kind of*. This relation generally produces a tree of classes, but it can also produce a directed acyclic graph (DAG).

The *depth* of an inheritance hierarchy is the length of the longest path from the root of the tree to a leaf.

The depth of the `Aircraft` hierarchy in Section 8 is 3. The depth of the `Expression` hierarchy in Section 9.1 is also 3, but this hierarchy is "wider" because some classes have siblings.

An inheritance hierarchy with many classes can be hard to understand. Depth tends to be more confusing than breadth, because the methods in a deep hierarchy can be executed at many different levels. Reiss says that a hierarchy with breadth should also have depth, but I have found useful applications of wide, flat hierarchies.

An inheritance hierarchy offers several advantages in program construction:

- Classes related by "is a kind of" usually have common features (a feature, as usual, is an attribute or method). Common features can be moved to (or, at least, towards) the

root of the tree so that they can be defined once only rather than once in each class. This is how inheritance supports *code reuse*.

- New code can be added to a program by declaring a new class that belongs to an inheritance hierarchy. This is how inheritance supports *extensibility*.
- We can write general code that applies to all members of a hierarchy. Furthermore, this code can contain "holes" that are filled by methods in derived classes.

  In Section 8, we defined `profit` = `revenue` − `expenses` in the base class `Aircraft`. An instance of a derived class, such as `Jumbo`, uses its own functions to compute `revenue` and `expenses`, and then uses the base class function to calculate `profit`.

9.4 Abstract Classes

An *abstract class* is a class that contains undefined methods. A class that contains no defined methods at all is sometimes called an *interface*. (Note that Java provides interfaces as a language construct. C++ does not, but we can consider a C++ class with pure virtual methods only to be an interface.)

A class that is not abstract is called *concrete*. That is, a concrete class contains no undefined methods. (This does not imply that a concrete class defines all of its methods, because it might inherit some or all of them.)

Some object oriented designers recommend that the internal classes of an inheritance hierarchy should be interfaces and that only the leaf classes should be concrete. While this approach has definite advantages, it is incompatible with code reuse and therefore removes one of the potential advantages of object oriented programming.

Suppose class `D` is derived from a base class `B`.

- If all of the methods of class `B` are pure virtual, we say that `D` inherits *behaviour* or *protocol* from `B`.
- If some of the methods of `B` are defined, we say that class `D` inherits *implementation* inheritance.

Most object oriented designers agree that behaviour inheritance is a good idea. There are some who feel that implementation inheritance is not a good idea and should be avoided. Reiss, for example, says on page 113: intermediate classes in a hierarchy should be abstract.

We have seen (in the principle of information hiding) that the interface of a class should be visible to clients whereas its implementation should not be. One way of achieving this is to declare an abstract class to serve as an interface and then derive a concrete class from it. This problem is specific to C++; it arises because a C++ class declaration must contain implementation information.

The main problem with this use of abstract classes is that the client does not have access to the implementation class and therefore has no way of constructing instances of it. The solution is to define yet another class, called a *factory class*, whose only purpose is to construct instances of the implementation class.

Here is a very simple example that illustrates the general idea. The goal is to provide the client with a class that solves the equation $ax = b$ but conceals the implementation of the

solution from the client. The client `#include`s the following class declarations. Note that these declarations do not reveal any implementation information.

```
class AbstractSolver
{
    public:
        virtual void setCoeffs (double a, double b) = 0;
        virtual void solve() = 0;
};

class Factory
{
    public:
        AbstractSolver *makeSolver();
};
```

The client could use these classes like this:

```
void main ()
{
    Factory fac;
    AbstractSolver *solver = fac.makeSolver();
    solver->setCoeffs(3.0, 5.0);
    solver->solve();
    delete solver;
}
```

Now we can reveal the mechanism that makes all of this work. First, we derive a concrete class from the abstract base class and provide implementations for its methods.

```
class ConcreteSolver : public AbstractSolver
{
    public:
        void setCoeffs (double a, double b);
        void solve();
    private:
        double d_a;
        double d_b;
};

void ConcreteSolver::setCoeffs (double a, double b)
{
    d_a = a;
    d_b = b;
}

void ConcreteSolver::solve ()
{
    cout << (d_b / d_a) << endl;
}
```

Next, we implement the factory class. It constructs a `ConcreteSolver` but returns a pointer to an `AbstractSolver`. The client, who does not see this implementation file, remains unawre of the existence of the class `ConcreteSolver`.

```
AbstractSolver *Factory::makeSolver ()
{
    return new ConcreteSolver;
}
```

An alternative technique for hiding implementation details is to use a *wrapper* class. This technique does not use inheritance. The client is provided with a "wrapper class" `Solver` that has the required methods and a pointer to a mysterious object:

```
class Solver
{
    public:
        Solver ();
        ~Solver ();
        void setCoeffs (double a, double b);
        void solve();
    private:
        Implementor *impl;
};
```

The wrapper class is used in the "obvious" way. The client does not know how `Solver` is implemented.

```
void main ()
{
    Solver solveIt;
    solveIt.setCoeffs(3, 5);
    solveIt.solve();
}
```

The supplier must provide a class `Implementor` that does the work. This class has the same methods as class `Solver`.

```
class Implementor
{
    public:
        void setCoeffs (double a, double b);
        void solve();
    private:
        double d_a;
        double d_b;
};

void Implementor::setCoeffs (double a, double b)
{
```

```
    d_a = a;
    d_b = b;
}

void Implementor::solve ()
{
    cout << (d_b / d_a) << endl;
}
```

The constructor for `Solver` constructs an `Implementor` on the heap. The destructor must destroy this instance.

```
Solver::Solver ()
{
    impl = new Implementor;
}

Solver::~Solver ()
{
    delete impl;
}
```

The methods of `Solver` simply call the corresponding methods of `Implementor`. Herein lies the inefficiency of the wrapper technique: each invocation of a method in `Solver` has to pass the call on to the `Implementor` object.

```
void Solver::setCoeffs (double a, double b)
{
    impl->setCoeffs(a, b);
}

void Solver::solve ()
{
    impl->solve();
}
```

An important advantage of both techniques — in practice, more important than the information hiding aspct — is that they allow changes of implementation without change of interface.

A good use of abstract base classes is to provide support for *multiple platforms*. The base class defines the interface that the platform must provide (e.g., windows, keystroke and mouse detection, etc.). One derived class is provided for each platform, and it implements all of the methods of the base class in a manner appropriate to that platform.

9.5 Miscellaneous Aspects of Inheritance

9.5.1 Access Control

C++ provides three access levels for derived classes. They are the same as access controls for variables: `public`, `protected`, and `private`.

- The most common (and recommended) form of inheritance is `public` inheritance, as in

  ```
  class D : public B { ....
  ```

  Access to inherited features is unchanged: a `public` feature of `B` is inherited as a `public` feature of `D`; similarly for `protected`; `private` features of `B` are, of course, not accessible in `D`. This is usually what is required.

- `private` inheritance is occasionally useful:

  ```
  class D : private B { ....
  ```

  The `public` and `protected` features of `B` can be used within `D` but are not accessible to clients of `D`.

  A typical application might be as follows: suppose that we have a class `Array` that implements arrays with automatic memory allocation, bounds checking, etc. We wish to implement a class `Stack`. The best way to do this is probably to introduce an instance of `Array` as an attribute of `Stack`. A alternative way is to use inheritance: `Stack` inherits `Array`.

  The problem with this is that we want clients to use only stack operations (`push()`, `pop()`, etc.) but, with inheritance, they will also be able to access the `Stack` as if it was an `Array`. The solution is `private` inheritance:

  ```
  class Stack : private Array { ....
  ```

  which allows us to use features of `Array` in `Stack` methods but does not make them accessible to clients of `Stack`.

  For most purposes, it is probably best to use `public` inheritance.

9.5.2 Multiple Inheritance

C++ allows a derived class to inherit from more than one base class. Although multiple inheritance should not be over-used, there are situations in which it is useful.

In many cases, what we actually want to is to inherit an implementation from one class and an interface from another class. (Although Java does not allow multiple inheritance, it does allow a class to inherit the implementation of one class and the interfaces of as many classes as necessary.)

An example might be as follows:

```
class Stack
{
// Defines a stack interface
....
};

class Array
{
```

```
// Implements an array
....
};

class List
{
// Implements a linked list
....
};

class StackWithArray : public Stack, private Array
{
// Implements a stack implemented with an array
....
};

class StackWithList : public Stack, private List
{
// Implements a stack implemented with a list
....
};
```

Although this would work, it would probably be simpler to do it without the implementation inheritance. The idea of inheriting the `Stack` interface *is* useful, because it enables us to write code that uses `Stack`s without committing to an implementation.

In practice, this kind of code would be written with templates, to provide `Stack<type>` for various `type`s.

9.5.3 Destructors

A base class should *always* have a virtual destructor. The dstructor will be redefined in the derived classes. To see the importance of this, consider a class hierarchy in which `B` is the base class and `D` is a typical derived class. We can write code like this:

```
B *pb = new D;
....
delete pb;
```

Since `pb` points to an instance of class `D`, the statement `delete pb` should call the destructor for class `D`. In order for this to happen, `B` must have a virtual destructor.

9.6 Avoiding Inheritance

Inheritance is not always the best solution. There are a number of circumatances where inheritance is a possible solution but not the best.

- If the differences between classes can be expressed entirely by giving different values to the attributes, there is probably no need to use inheritance.

In the `Aircraft` example of Section 8, many of the differences between classes could have been expressed simply by providing attributes such as `length`, `capacity`, `rate`, and so on. We could have solved the entire problem with a single class! The inheritance hierarchy would be justified only if distinct *calculations* are required in each class.

- If natural modelling requires that objects change their classes, inheritance may not be the best solution. It is possible to design systems in which classes change but it is not easy or natural.

  Hierarchies such as this one are popular in textbooks:

```
Person
    Student
        GraduateStudent
    Professor
        Assistant
        Associate
        Full
        Emeritus
```

  The problem with this organization is that a particular individual will migrate through various classes. We will have to implement special methods to convert a `GraduateStudent` into an `Assistant` (professor), etc. On the other hand, there does not appear to be much scope for making good use of inheritance.

10 Design Notation

If a system is simple enough, we can fit all the pieces together mentally before we start coding.

> "The first step in programming is imagining. Just making it crystal clear in my mind what is going thappen. Once I have the structure fairly firm and clear in my mind then I write the code." Charles Simonyi (Bravo, Word, Excel).

> "You have to simulate in your mind how the program is going to work." Bill Gates (BASIC).

> "I believe very strongly that one person . . . . should do the design and high-level structure. You get a consistency and elegance when it springs from one mind." John Page (PFS:FILE).

> "I can do anything that I can make an image of in my head." Jonathan Sachs (Lotus 1–2–3).

> "To do the kind of programming I do takes incredible concentration and focus . . . . keeping all the different connections in you brain at once." Andy Hertzfeld (MacOS)

Above a certain level of complexity, however, "having it all in your head" doesn't work. That level depends on the designer, and the designers quoted are clearly above average, but some systems are too complex for any one person to "have in the head". This implies that we must have a *notation* for design.

During the early 90s, there were a number of design methods and notations available. Four of the most prominent were conceived by:

- Grady Booch, who wrote *Object Oriented Analysis and Design with Applications* (1991, revised edition, 1994);
- James Rumbaugh, who designed Object Modelling Technique (OMT), first presented at OOPSLA in 1987; and
- Ivar Jacobson, who published *Object-Oriented Software Engineering* in 1992 and introduced "use cases" to object oriented software development.
- David Harel introduced *statecharts*.

Eventually, Rumbaugh and Jacobson joined Booch's company, Rational Software, and combined their work into a single notation called *Unified Modelling Language* or UML. Although the notation is both textual and visual, UML is best known for its diagrams, of which there are nine basic kinds.

Before introducing UML, a few comments are in order.

- A diagramming notation should be simple and precise. UML is quite complex: the "Notation Summary" occupies 60 pages and describes numerous symbols, keywords, and fonts. This is partly due to the origins of UML as a "union" of three separate notations.
- Because of its complexity, it is hard to use UML in a casual way. Like a programming language, you have to use it intensively for a significant period of time to become familiar with all of its details.
- Also because of its complexity, it is impractical to use UML as a hand-written notation. To use it effectively you need software that allows you to draw and modify diagrams and to check properties of diagrams.
- Much of UML is defined rather vaguely. There have been many attempts to provide a formal semantics, but none has been very succesful so far.
- There is a tremendous variety of software. It is unlikely that any *single* notation will be effective for all applications. Yet UML claims to be a universal notation, suited to all conceivable problems.
- Most of the enthusiasm about UML comes from higher management. People who have to use it are less enthusiastic about its capabilities.

The following notes are based mainly on *The Unified Modelling Language User Guide* by Grady Booch, James Rumbaugh, and Ivar Jacobson.

10.1 Modelling

According to the authors of UML, design is a *modelling* process. A language suitable for modelling is a *modelling language*.

Software design is the process of constructing conceptual models. UML is a language that designers can use to communicate their conceptual models to other designers, clients, and developers. UML provides facilities for *visualizing*, *specifying*, *constructing*, and *documenting*.

A UML model consist of:

Things are the abstractions that are first-class citizens in a model;

Relationships tie these things together; and

Diagrams group interesting collections of things.

Things come in the following varieties:

Structural Things correspond to nouns and include *classes*, *interfaces*, *collaborations*, *use cases*, *active classes*, *components*, and *nodes*;

Behavioural Things correspond to verbs and consist of *interactions* and *states*;

Grouping Things are the "organizational parts" of the model and consist mainly of *packages*;

Annotational Things allow you to describe, illuminate, and remark on aspects of the model and consist of *notes*.

Relationships are as follows:

Dependencies describe semantic relationships between things in which one thing may affect the semantics of another thing;

Associations describe connections between things such as aggregation (which relates a whole and its parts);

Generalizations describe parent/child or general/specific relationships;

Realizations describe relationships between classifiers (a classifier defines a contract that another classifier carries out).

Things and relationships appear in *diagrams*, of which there are nine kinds:

Class Diagrams show sets of related classes;

Object Diagrams show sets of related objects;

Use Case Diagrams show scenarios of a system;

Sequence Diagrams show how messages between related objects are ordered in time;

Collaboration Diagrams show how messages between related objects are structured, independently of time;

Statechart Diagrams show machines with states, transitions, events, and activities;

Activity Diagrams are statechart diagrams that show system functionality and control flow.

Component Diagrams show the static relationships between components of a system.

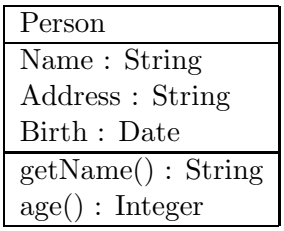
Deployment Diagram show the configuration of run-time nodes.

10.2 Class Diagrams

UML represents a class as a rectangular box with three fields:

- the name of the class;
- the attributes (data members) of the class;
- the methods (function members) of the class.

The fieds are separated by horizontal lines; the name field is required and the other fields may be omitted. An example:



UML uses Ada/Pascal syntax (*name colon type*) rather than C++ syntax (*type name*). If we are using C++ as the implementation language, this is confusing. Fortunately, the UML tools accept fairly arbitrary syntax, although the amount of checking we can do is limited if we do not use the standard syntax.

Classes can be *adorned* in various ways. For example, the name of a feature can be preceded by "+" (public), "#" (protected), or "−" (private) to represent its accessibility. The software tools provide icons that are more mnemonic than these symbols (simple diamond, diamond with lock and key, diamond with lock and no key).

Classes can be related in various ways:

- A solid line joining two classes indicates an *association* but does not show how the association is implemented.
- An arrow from class A to class B indicates that there is an asymmetric association between the classes: A can "find" B but B cannot "find" A. This could be implemented by, for example, A having a pointer to B.
- A dashed line between two classes indicates a *dependency* between the classes. Class A depends on class B if A cannot complete its tasks without B.
- A line with a diamond at the end indicates an *aggregation*. For example, we could draw a `Phonebook` containing a number of `Persons` by drawing a line between the two classes, with the diamond beside `Phonebook`.

- A line joining classes B and D with an open triangle at the end (pointing at B) indicates that D is derived from B.

The lines may be adorned in varous ways. Any association can be named simply by writing the name near the line. The number of objects involved ($1:1$, $1:n$, $m:n$,etc.) can be indicated by writing numbers at the end of the lines.

This much notation enables us to show how the classes of a design work together. Sometimes it is feasible to include all of the different kinds of relationships in a single diagram. If the system is at all complicated, however, it is probably better to put dependencies and associations in one diagram and inheritance (derivation) into another.

Class diagrams are *static*: they do not show how the system evolves with time. Some of the other kinds of diagram show the dynamic behaviour of the system.

A *message-trace* diagram shows how objects interact by passing messages to one another ("passing messages" is just OO jargon for "calling functions"). The horizontal axis shows the different objects involved and the vertical axis corresponds to time, moving down the page.

10.3 Assessment of Diagrams

Static diagrams:

- are useful for providing an overview of the system;
- can reveal problems such as high coupling;
- work best with less than 12 classes in a diagram.

Consequently, systems with more than 12 classes must be described by multiple diagrams. This reduces the usefulness of diagrams somewhat.

Diagrams without annotations convey very little.

Message trace diagrams handle a specific case in which only a small number of objects are involved. Consequently, we will require many diagrams for a complex system.

A major problem with design diagrams is that, after changes, they do not correspond to the software. What should happen:

- developer discovers a design error;
- design error is corrected and all affected design diagrams redrawn;
- developer modifies code according to revised design.

What usually happens in practice is that the code is altered, the design diagrams become out of step with the code, and are eventually discarded.

Rational uses the term **round-trip engineering** to describe a process whereby code can be generated automatically from design diagrams and, later, design diagrams can be "reverse engineered" from code. The technique depends on the fact that the code was generated by software tools; it cannot be applied to arbitrary code.

A problem with UML (and similar notations) is that there is a significant translation step from diagrams to code. UML designs cannot be executed, tested, or verified. It is possible (and advisable) to "walk through" UML designs.

Seamless software engineering attempts to avoid this translation by using a consistent notation throughout development. A seamless notation is usually textual (since one of its incarnations has to be the target programming language) but it may be possible to derive diagrams from the text.

11 Testing and Debugging

Testing is running a program and comparing actual results to expected results.

Debugging is finding the faults in a program that cause it to give unexpected results or to fail in some other way.

There are good reasons to avoid the term "bug". A bug sounds like something that flies in through the window and alights in your code. (Like the mythical bug that got trapped between the contacts of a relay in an early electric computer.) "Faults" are in the code because you put them there. It is your responsibility to take them out again.

Testing and debugging are related but different. In a large project, the testing teams will be distinct from the developers.

The problem with testing your own code is that you will tend to be too kind to it. A good tester is someone who tries as hard as possible to "break" the program. It is difficult to do that to your own creation! That is one reason why testing should not be done by the developers.

Debugging is detecting. The observed errors or behaviour provide clues as to what has gone wrong. You use those clues to discover faults in the program. You may find mistakes in the code, but this is not enough. You have corrected a fault only when you have shown that the coding errors that you found account for *all* of the observed behaviour. Having corrected the fault, you should also perform the tests again, of course.

The best way to avoid spending lots of time testing and debugging is to write correct code in the first place. This is easier said than done, but there are useful techniques.

- Keep everything as simple as possible (but no simpler!). Tricky code tends to contain more errors than simple code and is harder to debug as well.
- Minimize coupling. A fault in one component is easier to find than a fault that is spread over several components. If interfaces are simple and minimal, and interactions are restricted, there are fewer opportunities for this kind of fault to occur.
- When you design, think of all the things that might go wrong. If you can write the code in such a way that an error cannot possibly occur, do so. If this is not possible, include code that detects the error and responds appropriately.
- You can use *defensive coding* (put "sanity checks" in all your functions) or *contracts* (define the preconditions and postconditions of each function precisely). Contracts are better provided that they are respected. Preconditions, at least, should always be checked, at least during the testing phase.

- Use exception handling where it is appropriate. The appropriate time to use exception handling is when you cannot prevent an error situation arising by careful coding. Exceptions may occur during normal operation and the program must handle them gracefully.
- Use *assertions* where appropriate. The appropriate time to use assertions is to detect logical errors in the code — that is, situations that should never arise during normal operating conditions. When an assertion fails, the program crashes. It is therefore unacceptable for an assertion to fail in production software.
- Ensure that all data is correctly initialized. Initialization errors are a common source of obscure faults.
- Practice reviewing. Any artefact of the software process can be reviewed. Reviewing means sitting down with a group of people and going through a document (user manual, design, code, etc.) carefully and systematically.

  Reviewing is a more effective way of finding faults than debugging.

11.1 Assertions

The following code illustrates the use of assertions in C++.

```
#include <assert.h>
#include <math.h>

void main ()
{
    ....
    double z = ....
    assert(z >= 0);
    double qz = sqrt(z);
    ....
}
```

The program first calculates the value of `z`. If the calculation is correct, the result should be positive. The `assert` statement checks this: if $\texttt{z} \geq 0$, the `assert` statement has no effect but, if $\texttt{z} < 0$, the program will terminate immediately. If the following line is reached, we know that it is safe to calculate $\sqrt{z}$.

On most systems, the function `sqrt` will fail immediately if it is given a negative argument. (Another possibility is that it might raise an exception.) Given this, we might question the value of the `assert` statement: the program is going to fail anyway, why bother?

There are two answers to this question:

- The `assert` statement provides important documentation: it announces clearly that the programmer believes that $\texttt{z} \geq 0$ at that point in the program.
- The error message generated by the `assert` statement will almost certainly be more helpful than the error message generated by `sqrt`.

  When `sqrt` fails, you will probably get a message like this:

```
sqrt: domain error
```

When `assert` fails, the message will be more like this:

```
Assertion failed: z >= 0.0, file calc.cpp, line 87
```

Other common uses of assertions are to ensure that:

- numbers are within range: `assert(0 <= level && level < MAX);`;
- pointers are not `NULL`: `assert(ptr);`;
- strings are not empty: `assert(strlen(message) > 0);`.

You can disable the effect of assertions by compiling with the flag `NDEBUG` set.

11.2 Exceptions

We use `assert` statements to check the logical consistency of the program. When an assertion fails, we know that there must be a fault in our reasoning about how the program is supposed to work.

There are other kinds of error that we cannot prevent by reasoning. Suppose, for example, that we are inverting a large matrix which might (with low probability) be singular. If the matrix is singular (its determinant is zero), there will be an attempt to divide by zero. We would like to report failure without aborting the program. There are two ways in which we might do this:

- ensure that the matrix is non-singular before starting to invert it;
- ensure that every divisor is non-zero.

Both of these alternatives are inefficient. Testing for singularity requires finding the determinant of the matrix, which has the same time complexity as inverting it. Checking every division is wasteful since we know that a zero divisor is unlikely.

The solution is to use *exception handling*. "Exceptions" are objects constructed (or "generated") when something goes wrong. For example, division by zero generates an exception. Exceptions are "thrown" from the problem area and "caught" by an *exception handler*.

Exceptions provide a form of one-way communication between two parts of the program. For example, an exception might be thrown by a library function and caught by an application. This is useful because a library function that detects an error does not have enough information to process the error in a sensible way (consider `sqrt` above, for instance).

The syntax for exception handling in C++ introduces three new keywords: `throw`, `try`, and `catch`. We use `throw` when we detect the error:

```
double sqrt (double x)
{
    if (x < 0.0)
        throw "sqrt: negative argument"
    ....
}
```

and we use `try` and `catch` to process exceptions:

```
....
try
{
    // (A)
}
catch (char *message)
{
    // (B)
}
// (C)
....
```

The code in block (A) contains one or more calls to `sqrt`. In general, this code may call functions, that call functions, . . . that call `sqrt`. If this code executes normally (no negative arguments for `sqrt`) during (A), then the program continues at (C).

If at any point during the execution of (A), the program calls `sqrt` with a negative argument, control jumps to the exception handler, which is block (B). The code in block (B) responds to the error in an appropriate way. For example, it might display an error message, abandon the current task and start another one, or even terminate the program. Within block (B), the variable `message` has the value `"sqrt:  negative argument"`. In the general case, the exception handler is provided with the exception object that was thrown. In this example, the object is a string, but it doesn't have to be.

Note:

- It is helpful to think of `catch (char *message)` as a kind of function header. The string `"sqrt:  negative argument"` is like an argument passed to the function. When the exception is an object (examples below), it is passed by reference to avoid copying.
- Although the exception handling is simple to use, there is a lot going on "behind the scenes". When `throw` is called, the program may have called several functions and entered several scopes. The program must exit from the functions and scopes correctly (this will often involve calling destructors) before handling the exception. All of this is done automatically, by the system, so the programmer does not have to worry about it.

Here is an example of exception handling with exception objects. The first class is a base class for errors that can occur while driving.

```
class CarErrors
{
public:
    CarErrors (char *name) { d_name = name; }
private:
    char *d_name;
};
```

The next two classes are derived from the base class. The first is for speeding errors and it has an attribute for the speed of the car. In general, attributes of exception objects can be used to store useful information about the error.

```
class SpeedingError : public CarErrors
{
public:
    SpeedingError (char *name, int speed);
    int getSpeed () { return d_speed; }
private:
    int d_speed;
};

SpeedingError::SpeedingError (char *name, int speed)
    : CarErrors(name)
{
    d_speed = speed;
}
```

The second derived class is for crashes; it does not contain any additional data.

```
class CrashError : public CarErrors
{
public:
    CrashError (char *name) : CarErrors(name) {}
};
```

The next class describes the cars that will be speeding and crashing.

```
class Car
{
public:
    Car () { d_speed = 0; d_name = "Volvo"; }
    void accelerate ();
    void turn () throw (CrashError);
private:
    char *d_name;
    int d_speed;
};
```

The methods for the car throw exceptions. The exception objects are obtained by calling constructors of the exception classes.

```
void Car::accelerate ()
{
    d_speed += 10;
    if (d_speed > 100)
        throw SpeedingError(d_name, d_speed);
    cout << "Speed = " << d_speed << endl;
}

void Car::turn () throw (CrashError)
{
```

```
    if (d_speed > 80)
        throw CrashError(d_name);
    cout << "Turning" << endl;
}
```

The declaration of `turn()` contains the expression `throw (CrashError)`; this is an *exception specification*. It is part of the type of the function and must be included in both the declaration and the definition. It is recommended but not required (note that `accelerate()` does not have an exception specification). Exception specifications provide useful documentation: someone reading the declaration of class `Car` can see immediately that `turn()` may throw `CrashError`.

Here is a simple program that uses class `Car`.

```
void main ()
{
    Car clunker;
    try
    {
        for (int n = 0; n < 50; n++)
        {
            if (random(10) > 3)
                clunker.accelerate();
            else
                clunker.turn();
        }
    }
    catch (SpeedingError se)
    {
        cout << "Speeding at " << se.getSpeed() << endl;
    }
    catch (CrashError ce)
    {
        cout << "Crashed!" << endl;
    }
}
```

When this program runs, the code in the `try` block invokes the methods `accelerate()` and `turn()` in a random order. Eventually, one of them throws an exception. The exception is caught by the `catch` block, the program prints an appropriate error message, and stops.

Why did we construct a hierarchy of error classes, instead of just having two independent classes? The hierarchy allows us to distinguish between specific errors and general errors. In the main program, we could replace the two `catch` blocks by:

```
catch (CarErrors c)
{
    cout << "Something went wrong with the car." << endl;
}
```

This block will handle `SpeedingError`s and `CrashError`s. The trade-off is that we can no longer access the specific information provided by these errors (e.g., the speed of a `SpeedingError`).

The system tries to match `catch` blocks in the order that they appear. Suppose that there are many class derived from `CarErrors`. The following handler will perform one action if the car crashes and another action if *any other* exception is raised:

```
catch (CrashError ce)
{
    // Action for crashes
}
catch (CarErrors c)
{
    // Action for any other problem
}
```

There is also a special form of the `catch` statement that will respond to *any* exception:

```
catch (...)
{
    cout << "An exception was raised." << endl;
}
```

A `catch` block may contain the statement `throw` with no arguments. The effect is to raise the same exception and pass it further up the call chain. In the following example, the cop gives a speeding ticket to a speeding car but, if the car is going at more than 150 kph, passes control to a higher authority:

```
catch (SpeedingError se)
{
    if (se.getSpeed() <= 150)
        // issue speeding ticket
    else
        throw;
}
```

If a program raises an exception for which there is no matching handler, it is terminated. More precisely, the system calls the default version of the function `terminate()`, which terminates the program. It is possible to provide your own definition of `terminate()` if you want some other kind of behaviour.

The C++ library has a collection of *standard exception classes* that are arranged in a hierarchy as follows:

```
exception
      logic_error
            domain_error
            invalid_argument
            length_error
            out_of_range
      runtime_error
            range_error
            overflow_error
```

```
bad_alloc
bad_cast
bad_typeid
bad_exception
```

You can use these classes to handle errors that would normally cause the program to crash. The function `what()` returns a string (`char *`); it is defined in the base class `exception` and redefined in the derived classes.

```
#include <stdexcept>
....
try
{
    .... log(x) ....
}
catch (range_error re)
{
    cout << re.what() << endl;
}
```

There are some further details concerned with exception handling, but the uses described above are adequate for most applications

11.3 Debugging Tools

Most software developments come with debugging tools. There are several Software Development Environments (SDEs) that run under Windows, such as Visual C++ and Borland C++. These have debugging tools built into them. SDEs are available for UNIX/Linux, but they are usually expensive packages intended for professional programmers. There are standalone debuggers for UNIX and Linux, however.

A good debugger should allow you to work at the level of source code. The debugger should create the illusion that your program is executed in steps, with each step corresponding to one line of source code. You should be able to step through the program in various ways, allow it to run to a breakpoint, and check the values of variables at any point.

Using a debugger requires practice: it is possible to waste a lot of time using a debugger ineffectively. Although some people advocate stepping through your program one line at a time, this is not feasible for programs of any complexity.

The best approach to debugging is scientific. Your program corresponds to *nature*. You are a *scientist* who can make *observations*. Rather than making random observations without a plan, you should formulate a *theory* that accounts for the observed behaviour of your program. You should then conduct *experiments* designed to *test* the theory.

(Note that this analogy applies also to testing. In this case the theory is that *the program works according to its specification*. A test is an experiment intended to refute the theory — that is, to find a situation in which the program does not work.)

There are a number of advanced debugging tools that perform sophisticated analysis of the program, either statically or during execution. These tools are usually too expensive for private use but are extensively used in the software industry. They are most useful for detecting memory management problems: dangling pointers, leaks, and so on.

If you don't have a debugging tool, the best way to debug is to introduce print statements into the program so that you can see what it is doing. Here is a technique that I use myself:

- Declare an output file stream with a name like "`log.txt`". This is called the "log file".
- Ensure that the log file is accessible from all parts of the program that are under test (i.e., it is a global variable!).
- Add output statements that write the values of significant data items to the log file.
- After each run, read the log file. If an output statement is printing *exactly* what you expect it to print, delete it.
- If an output statement is printing something that looks odd, investigate the cause.
- If all output statements are giving expected results, and the program still isn't working, add some more output statements.

12 A Sample Program

The program that we study in this section is called `Parseval`. It is in fact not intended to be a complete program but rather a reusable component that can be used in other programs. It provides a single class, called `Parseval`, that can parse and evaluate expressions.

The following code is a simple test for `Parseval`. It doesn't do much but it shows how `Parseval` is used.

```
#include "parseval.h"
#include <iostream.h>
#include <string.h>

void main ()
{
    Parseval *p = new Parseval();
    while (true)
    {
        char expr[100];
        cout << "\nEnter expression (0 to quit): ";
        cin >> expr;
        if (strcmp(expr, "0") == 0)
            break;
        bool ok;

        p->parse(expr, ok);

        if (ok)
        {
            bool playing = true;
            while (playing)
            {
```

```
                char c;
                char name[100];
                double value;
                cout << "(e)val, (s)et, or (n)ew expression? ";
                cin >> c;
                switch (c)
                {
                case 'e':
                    cout << (p->eval()) << endl;
                    break;
                case 's':
                    cout << "Variable? ";
                    cin >> name;
                    cout << "Value?    ";
                    cin >> value;
                    p->set(name, value);
                    break;
                default:
                    playing = false;
                    break;
                }
            }
        }
    }
}
```

The program starts by constructing a new instance of `Parseval`. The rest of it consists of a loop in which the following events occur. First, the program asks the user for an expression. The expression is entered as a string in conventional algebraic format — or as close to it as ASCII permits. Here are some examples of valid expressions.

```
x^2 - 3*x + 5
(x - y)^2 / (x + y)^2
3*sin(x) + 5.5*cos(x)
|x|^2
```

Note that `Parseval` accepts the usual operators (`+`, `-`, `*`, `/`, and `^` (for exponents) as well as `|` (for absolute value).

`Parseval` is called to parse the expression; it sets the flag `ok` to indicate whether parsing succeeded. If it was successful, the user is offered the option of evaluating the expression, setting values of the variables in the expression, or entering a new expression. Note that all of `Parseval`'s capabilities are expressed through three functions: `parse()`, `eval()`, and `set()`.

Here is the declaration for the class `Parseval`.

```
class Parseval
{
public:
```

```
    Parseval ();
    ~Parseval ();

    void parse (char *expression, bool & ok);
        // Attempt to parse the given string, and set 'ok'
        // to 'true' if successful, 'false' otherwise.

    double eval ();
        // Evaluate the expression using current values of variables.

    void set (char *name, double value) { psc->set(name, value); }
        // Set value of variable 'name' to 'value'.

private:
    // This group of functions parse various forms of expression.
    Node *parseExpr (Scanner *psc);
    Node *parseTerm (Scanner *psc);
    Node *parseFactor (Scanner *psc);
    Node *parsePrimary (Scanner *psc);

    Scanner *psc;
    Node *expr;
};
```

`Parseval` represents expressions as trees consisting of nodes — instances of class `Node`. `Node` is an abstract base class with only two virtual functions: a destructor that does nothing and a pure virtual function `eval()`, which evaluates the tree rooted at this node.

```
class Node
{
public:
    virtual ~Node() {};
    virtual double eval() = 0;
};
```

Classes derived from `Node` describe particular kinds of expression. The class `Number` contains a constant value. It relies on the default destructor because it does not have any pointers.

```
class Number : public Node
{
public:
    Number (double ivalue) { value = ivalue; }
    double eval () { return value; }
private:
    double value;
};
```

The next class derived from `Node` is `Variable`; an instance represents a variable in an expression. Variables may have values assigned to them, and we want a variable to have the same value wherever it appears in the expression. Consequently, a variable node contains a pointer to a table of variable names and values. Class `Variable` uses the default destructor. Note that it would be a mistake for a `Variable` to delete the table entry because it doesn't own the table.

```
class Variable : public Node
{
public:
    Variable (Entry *ientry) { entry = ientry; }
    double eval () { return entry->eval(); }
private:
    Entry *entry;
};
```

Binary operators in the expression are represented by the class `Binary`. An instance has an enumerated value (defined in class `Scanner` which we haven't see yet) and pointers to its left and right subtrees. Its destructor deletes these subtrees.

```
class Binary : public Node
{
public:
    Binary(Scanner::KIND ikind, Node *ileft, Node *iright);

    ~Binary ()
    {
        delete left;
        delete right;
    }

    double eval();

private:
    Scanner::KIND kind;     // Determines the operator.
    Node *left;             // First operand.
    Node *right;            // Second operand.
};
```

The constructor for instances of `Binary` simply copies its arguments.

```
Binary::Binary (Scanner::KIND ikind, Node *ileft, Node *iright)
{
    kind = ikind;
    left = ileft;
    right = iright;
}
```

The evaluator for binary nodes evaluates its subtrees first and then applies the operator at the root. Division by zero is checked, beacuse it's easy, but the function does not check the arguments of `pow()` when evaluating x^y.

```
double Binary::eval ()
{
    double lval = left->eval();
    double rval = right->eval();
    switch (kind)
    {
        case Scanner::ADD:
            return lval + rval;
        case Scanner::SUB:
            return lval - rval;
        case Scanner::MUL:
            return lval * rval;
        case Scanner::DIV:
            if (rval == 0.0)
                throw(new Error("div", 0));
            else
                return lval / rval;
        case Scanner::EXP:
                return pow(lval, rval);
        default:
            return 0.0;
    }
}
```

Finally, we need a class for nodes that represent function calls in the expression. The class `Function` is derived from `Node`. It has a pointer to an instance of `Absfun` (described next) and a pointer to its argument. The destructor deletes the argument but not the `Absfun`.

```
class Function : public Node
{
public:
    Function (char *name, Node *iarg, bool & defined);
    ~Function ()
    {
        delete arg;
    }

    double eval ();

private:
    Absfun *fun;    // Pointer to function object.
    Node *arg;      // Pointer to argument expression.
};
```

The constructor for a function attepts to match the function name with the name of one of the stored functions. If it doesn't succeed, it reports failure. Note that this search for the function takes place during *parsing*: there is no overhead involved in finding a function during *evaluation*.

```
Function::Function (char *name, Node *iarg, bool & defined)
{
    for (int i = 0; functions[i] != NULL; i++)
    {
        if (strcmp(name, functions[i]->name) == 0)
        {
            fun = functions[i];
            arg = iarg;
            defined = true;
            return;
        }
    }
    defined = false;
    fun = NULL;
    arg = NULL;
}
```

When a function call is evaluated, the argument for the function is evaluated and the result is passed to the function.

```
double Function::eval ()
{
    return fun->eval(arg->eval());
}
```

The class `Absfun` is an abstract base class for all functions that are recognized. Although it has a pointer to the name of the function, this name is not allocated by function objects; consequently, `Absfun` does not need a special destructor.

```
class Absfun
{
public:
    virtual double eval (double x) = 0;
    void store (char *iName);
public:
    char *name;
};

void Absfun::store (char *iName)
{
    name = new char[strlen(iName) + 1];
    strcpy(name, iName);
}
```

For each function that we want `Parseval` to recognize, we declare a class. The class has a constructor that stores the name of the function in the object and a function `eval()` that evaluates the functions when called. If the function can fail (e.g., `sqrt(x)` with `x` < 0), then the function throws an instance of class `Error`. Here are some typical functions.

```
class Sin : public Absfun
{
public:
    Sin::Sin () { store("sin"); }
    double eval (double x) { return sin(x); }
};

class Cos : public Absfun
{
public:
    Cos::Cos () { store("cos"); }
    double eval (double x) { return cos(x); }
};

class Tan : public Absfun
{
public:
    Tan::Tan () { store("tan"); }
    double eval (double x) { return tan(x); }
};

class Atan : public Absfun
{
public:
    Atan::Atan () { store("atan"); }
    double eval (double x) { return atan(x); }
};

class Exp : public Absfun
{
public:
    Exp::Exp () { store("exp"); }
    double eval (double x) { return exp(x); }
};

class Log : public Absfun
{
public:
    Log::Log () { store("log"); }
    double eval (double x)
    {
        if (x <= 0.0)
            throw(new Error("log", x));
```

```
        else
            return log(x);
    }
};

class Sqrt : public Absfun
{
public:
    Sqrt::Sqrt () { store("sqrt"); }
    double eval (double x)
    {
        if (x < 0.0)
            throw(new Error("sqrt", x));
        else
            return sqrt(x);
    }
};

class Fabs : public Absfun
{
public:
    Fabs::Fabs () { store("fabs"); }
    double eval (double x) { return fabs(x); }
};

class Step : public Absfun
{
public:
    Step::Step () { store("step"); }
    double eval (double x) { return x < 0 ? 0.0 : 1.0; }
};
```

There is a global array of pointers to function objects, defined like this:

```
Absfun *functions[] =
{
    new Sin,
    new Cos,
    new Tan,
    new Atan,
    new Exp,
    new Log,
    new Sqrt,
    new Fabs,
    new Step,
    NULL        // Do not remove!  (Needed for loop termination.)
};
```

`Parseval` has to parse expressions. It uses a *scanner* to handle the low-level details of reading.

The scanner reads characters from the string that is passed to it, and constructs *tokens* that represent parts of the expression. For example, `sin` is a single token. The scanner is also responsible for skipping blanks and recognizing the end of the string.

The enumeration `KIND` describes the various kinds of token that the scanner recognizes. In principle, we could introduce an abstract base class `Token` with derived classes for the different kinds of token. In practice, this is hardly worth the effort. Instead, the scanner stores information about the current token and provides inquiry functions so that clients can obtain the information they need about the token. There are some security leaks here: for example, it does not make sense to call `getEntry()` if the token `kind` is `NUM`. However, nothing prevents the client from doing this.

The scanner manages the symbol table, which is a binary search tree. The attribute `root` points to the root of this tree.

```
class Scanner
{
public:
    enum KIND
        {
            BAD, EOS, VAR, NUM, ADD, SUB,
            MUL, DIV, EXP, LP, RP, ABS
        };

    enum { BUFLEN = 120 };  // Size of temporary buffer.

    Scanner(char *itext);

    ~Scanner ()
    {
        delete root;
    }

    void next();
        //  Scan one token of the input string.

    KIND getKind () { return kind; }
    Entry *getEntry () { return entry; }
    double getValue () { return numval; }

    void set (char *name, double value)
    {
        helpSet(name, value, root);
    }

    void error (char *message);
    bool ok () { return success; }

private:
```

```
    Entry *enter(char *name, Entry * & root);
        // Enter a variable name into the symbol table.

    void helpSet (char *name, double value, Entry *root);
        // Recursive function to set a value.

    Entry *root;            // Symbol table.

    // Contents of current token.
    KIND kind;
    Entry *entry;           // Symbol table entry of variable.
    double numval;          // Value if number.
    char *text;             // Text to scan.
    char *pcb;              // Current character.
    char buffer[BUFLEN];    // For variables and numbers.
    bool success;           // No errors have occurred.
};
```

The constructor for class `Scanner` keeps the pointer to the expression passed to it (it doesn't need to copy the expression) and initializes the attributes. In particular, the symbol table pointer is set to `NULL`.

```
Scanner::Scanner (char *itext)
    // Construct a scanner for the give input text.
{
    text = itext;
    pcb = text;
    success = true;
    root = NULL;
    next();
}
```

The most complicated function in the scanner is `next()`, which finds the next token. It skips blanks and table, checks for the end of the string, and then classifies the token according to the non-blank characters that it reads.

```
void Scanner::next()
{
    char *q;
    while (*pcb == ' ' || *pcb == '\t')
        pcb++;
    if (*pcb == '\0')
        kind = EOS;

    // Check for names
    else if (isalpha(*pcb))
    {
        q = buffer;
```

```
        while (isalnum(*pcb))
            *q++ = *pcb++;
        *q = '\0';
        entry = enter(buffer, root);
        kind = VAR;
    }

    // Check for:
    //   { digit } [ '.' { digit } ]
    //   [ ( 'e' | 'E' ) [ '+' | '-' ] { digit } ]
    else if (isdigit(*pcb) || *pcb == '.')
    {
        q = buffer;
        while (isdigit(*pcb))
            *q++ = *pcb++;
        if (*pcb == '.')
        {
            *q++ = *pcb++;
            while (isdigit(*pcb))
                *q++ = *pcb++;
        }
        if (*pcb == 'e' || *pcb == 'E')
        {
            *q++ = *pcb++;
            if (*pcb == '+' || *pcb == '-')
                *q++ = *pcb++;
            while (isdigit(*pcb))
                *q++ = *pcb++;
        }
        *q = '\0';
        char *r;    // Character at which strtod stopped.
        numval = strtod(buffer, &r);
        if (r < q)
            error("Error in number");
        kind = NUM;
    }
    else
    {
        // Check for other characters.
        switch (*pcb++)
        {
            case '+':
                kind = ADD;
                break;
            case '-':
                kind = SUB;
                break;
            case '*':
```

```
                kind = MUL;
                break;
            case '/':
                kind = DIV;
                break;
            case '^':
                kind = EXP;
                break;
            case '(':
                kind = LP;
                break;
            case ')':
                kind = RP;
                break;
            case '|':
                kind = ABS;
                break;
            default:
                error("Illegal character");
                kind = BAD;
                break;
        }
    }
}
```

The function `Enter()` makes an entry in the symbol table. Note that the argument `root` is passed as "reference to (pointer to `Entry`)". This is because we are passing a pointer to the function, but it may need to change the value of the pointer. This is the standard technique for recursively adding a new entry to a binary search tree.

```
Entry *Scanner::enter (char *name, Entry * & root)
{
    if (root == NULL)
    {
        root = new Entry(name);
        return root;
    }
    else
    {
        int cmp = strcmp(name, root->getName());
        if (cmp == 0)
            return root;
        else if (cmp < 0)
            return enter(name, root->left);
        else // cmp > 0
            return enter(name, root->right);
    }
}
```

The scanner provides an inlined function `set()` to set the value of a variable in the symbol table. The name of the variable and the desired value are passed to it. The function `helpSet()` is a "helper" function used to implement `set()`: it is essentially the same but has an extra argument to help with the recursion.

```
void Scanner::helpSet (char *name, double value, Entry *root)
    // Set the value of a variable in the symbol table.
    // If the variable does not exist, do nothing.
{
    if (root)
    {
        int cmp = strcmp(name, root->getName());
        if (cmp == 0)
            root->set(value);
        else if (cmp < 0)
            helpSet(name, value, root->left);
        else // cmp > 0
            helpSet(name, value, root->right);
    }
}
```

Error handling is the most complicated aspect of scanning and parsing. The problem is that, after encountering an error, the scanner doesn't really know what to do next because it is "outof step" with the input. The solution adopted here is rather simple: we copy the characters scanned so far so that the user knows where the error occurred, and display them with a hopefully informative message. Errors after the first error are not reported because they are most likely to be caused by the first error.

```
void Scanner::error (char *message)
{
    if (success)
    {
        ostrstream os;
        char *p = text;
        while (p <= pcb)
        {
            if (*p != '\0')
                os << *p;
            p++;
        }
        os << "  " << message << '.' << ends;
        cerr << os.str();   // Change for Windows
        delete [] os.str();
        success = false;
    }
}
```

The symbol table is not stored as such. Instead, it is represented by instance of class `Entry`.

Each instance has a name, a value, and pointers to its left and right subtrees. These pointers are `public` (this simplifies the code slightly) although they should probably be public.

```
class Entry
{
public:
    Entry (char *iname);
    ~Entry ();

    double eval () { return value; }
    char *getName () { return name; }
    void set (double ivalue) { value = ivalue; }

    // The pointers to subtrees are public
    // to simplify access by 'enter'.
    Entry *left;
    Entry *right;

private:
    char *name;     // Variable name.
    double value;   // Value.
};
```

The constructor for `Entry` makes its own copy of the name to avoid sharing problems. The destructor deletes both the name and the subtrees of the entry.

```
Entry::Entry (char *iname)
{
    name = new char[strlen(iname) + 1];
    strcpy(name, iname);
    value = 0.0;
    left = NULL;
    right = NULL;
}

Entry::~Entry ()
{
    delete [] name;
    delete left;
    delete right;
}
```

Instances of the class `Error` are exceptions thrown by the scanner or evaluator. Each error has a "name" which is a short descriptive string and a value.

```
class Error
{
public:
```

```
    Error (char *iname, double ival)
    {
        name = iname;
        value = ival;
    }

    void report();

private:
    char *name;
    double value;
};
```

The form of the reporting function for `Error`. The version shown here sends a message to the standard error stream. A Windows version might display a `MessageBox`.

```
void Error::report ()
{
    ostrstream os;
    os << "Domain error: " << name << '(' << value << ')' << ends;
    cerr << endl << os.str() << endl;
    delete [] os.str();
}
```

The last step is to implement the functions of class `Parseval`. The constructor and destructor are straightforward.

```
Parseval::Parseval ()
{
    psc = NULL;
    expr = NULL;
}

Parseval::~Parseval ()
{
    delete psc;
    delete expr;
}
```

The function `eval()` uses esception handling. It attempts to evaluate the expression but, if anything goes wrong, it catches the `Error` exception object, calls its `report` function, deletes it, and returns zero.

```
double Parseval::eval ()
    // Attempt to evaluate the expression.
    // Handle and report exceptions.
{
    try
```

```
    {
        return expr->eval();
    }
    catch (Error *perr)
    {
        perr->report();
        delete perr;
        return 0.0;
    }
}
```

The most complicated function of `Parseval` is the parser. It uses a technique known as *recursive descent*, based on a collection of mutually recursive functions that match the structure of expressions.

The top level function deletes any old scanners and expressions that are lying about and tries to parse the expression it is given. It class `parseExpr()` to do this and, when this function returns, checks that it did actually reach the end of the string. Without this check, the parser would not report an error after reading a string such as "`4*(x+y))`".

```
void Parseval::parse (char *expression, bool & ok)
{
    delete psc;
    psc = new Scanner(expression);
    delete expr;
    expr = parseExpr(psc);
    if (psc->getKind() != Scanner::EOS)
        psc->error("Unexpected character");
    ok = psc->ok();
}
```

The function `parseExpr()` looks for *sums of terms*, where "sums" includes both + and −.

```
Node *Parseval::parseExpr (Scanner *psc)
    // Parse Expr -> [ '-' ] Term { ( '+' | '-' ) Term }.
{
    Node *left;
    if (psc->getKind() == Scanner::SUB)
    {
        psc->next();
        left = new Binary(Scanner::SUB, new Number(0.0), parseTerm(psc));
    }
    else
        left = parseTerm(psc);
    while (  psc->getKind() == Scanner::ADD ||
             psc->getKind() == Scanner::SUB)
    {
        Scanner::KIND k = psc->getKind();
        psc->next();
```

```
        left = new Binary(k, left, parseTerm(psc));
    }
    return left;
}
```

The function `parseTerm()` tries to parse terms that consist of *products of factors*, where "products" includes $\times$ and $\div$.

```
Node *Parseval::parseTerm (Scanner *psc)
    // Parse Term -> Factor { ( '*' | '/' ) Factor }.
{
    Node *left = parseFactor(psc);
    while (  psc->getKind() == Scanner::MUL ||
             psc->getKind() == Scanner::DIV)
    {
        Scanner::KIND k = psc->getKind();
        psc->next();
        left = new Binary(k, left, parseFactor(psc));
    }
    return left;
}
```

A factor is a *primary* (see below) or, possibly, a primary with an exponent which is also a primary.

```
Node *Parseval::parseFactor (Scanner *psc)
    // Parse Factor -> Primary [ '^' Primary ].
{
    Node *left = parsePrimary(psc);
    if (psc->getKind() == Scanner::EXP)
    {
        psc->next();
        left = new Binary(Scanner::EXP, left, parsePrimary(psc));
    }
    return left;
}
```

A primary can be a number, a variable, an expression in parentheses, or an absolute value expression (e.g., `|x-y|`). Since a "variable" may turn out to be a function name, this function looks for "`(`" following a name. If a function call is found, a new function is constructed with the parsed argument.

```
Node *Parseval::parsePrimary (Scanner *psc)
    // Parse Primary -> NUM | VAR [ '(' Expr ')' ] | '(' Expr ')'.
{
    double value;
    bool defined;
    Entry *entry;
```

```
Node *result;
switch (psc->getKind())
{
    case Scanner::NUM:
        value = psc->getValue();
        psc->next();
        return new Number(value);

    case Scanner::VAR:
        // Variable or function application.
        entry = psc->getEntry();
        psc->next();
        if (psc->getKind() == Scanner::LP)
        {
            // Function application.
            psc->next();
            result = new Function(entry->getName(),
                                  parseExpr(psc), defined);
            if ( ! defined)
                psc->error("Undefined function");
            if (psc->getKind() == Scanner::RP)
                psc->next();
            else
                psc->error("')' expected");
            return result;
        }
        else
            return new Variable(entry);

    case Scanner::LP:
        psc->next();
        result = parseExpr(psc);
        if (psc->getKind() == Scanner::RP)
            psc->next();
        else
            psc->error("')' expected");
        return result;

    case Scanner::ABS:
        psc->next();
        result = new Function(FABS, parseExpr(psc), defined);
        if ( ! defined)
            psc->error("Undefined function");
        if (psc->getKind() == Scanner::ABS)
            psc->next();
        else
            psc->error("'|' expected");
        return result;
```

```
        default:
            psc->error("Syntax error");
            return NULL;
    }
}
```

13 User Interface Design

The most important issues in user interface design (UID) are aesthetic and ergonomic. "Aesthetic" (or "esthetic") means that the screen should be attractive to look at. "Ergonomic" means that the interface should be easy to use. A ideal UI is invisible: the user is not aware of using it. Very few computer programs reach this level.

The aesthetic and ergonomic aspects of UID are beyond the scope of this course. (They are covered in detail in COMP 675 *User Interface Design.*) In this section, we discuss how object oriented methods affect UID.

An important transition in UID occurred with the introduction of windows and mice. The earliest systems of this kind were developed at Xerox's Palo Alto Research Center (PARC) in the late 70s. They were taken up by Apple and later by Microsoft and are now ubiquitous.

Before the windows/mouse interface appeared, interactions were controlled by the program. You (the user) would either be told to perform an action (e.g., fill in a text or number field) or asked to select from a small number of options. When you filled in a form, you had to complete the fields in an order determined by the program, although a "flexible" system might allow you to go back to the previous field or forward to the next field.

The windows/mouse interface gives the user much more control. There are typically several windows on the screen, not necessarily belonging to the same program. The windows have various "controls" and areas where text can be entered. You can move windows, resize them, hide or restore them, click on any control, write text in the text areas, and so on. Sometimes the program will indicate things that you cannot do, for example by "greying" controls, but generally you have a much greater freedom than with the older interfaces.

This transfer of control required a corresponding change in program design. The old style programs had a procedural design with input and output statements in places convnenient for the programmer. The program executed statements in a convenient sequence, asking the user for input when necessary.

In the new style of programming, the programmer must assume that the user can do anything at any time. User actions are called *emph* and the program is a collection of *event handlers* that respond to the user's actions.

Windows libraries conceal many of the details from the application programmer. For example, the program does not have to know that a window has been moved. When the user resizes a window, the window library displays the new window frame and sends a "resize" message (which includes the new dimensions of the window) to the program: the programmer's responsibility is simply to redisplay the contents of the window taking into account the window's new size.

With object oriented programming, it is possible to make all of this very natural. There is an object in the program corresponding to each window on the screen. The window manager

captures all mouse and keyboard events and sends them to the appropriate object. Thus objects will have methods that correspond to operations such as "resize", "scroll up", and "display character".

Although the earliest systems (Smalltalk and others at PARC) were built in this way, subsequent systems were not. This is mainly because the windows environments developed before object oriented languages were in widespread use. For example, the X window system was developed at MIT using C, as was Motif, a toolkit that runs on top of X windows. Similarly, Microsoft the Windows systems were developed first in Pascal and later in C. The Microsoft Foundation Classes (MFC) *are* object oriented (they are written in C++) but they run on top of the windows code in Windows 95/98/NT/2000.

13.1 Coding for Windows

When you design a UI, your goal should be to give the user as much control as possible. You decide what is possible (that is, what controls you will provide and what events you will handle) and then design the code in such a way that any event can be processed at any time. If the user operates controls incorrectly, the program should respond in a friendly and helpful way. For example, if the user clicks `Next` before required fields have been completed, the program should display something like "`Please complete fields X, Y, Z before proceeding`".

A common mistake in UI programming is to assume that, because events can occur in any order, there must be a lot of global data. It is not always easy to combine information hiding with good UI design, but it is possible. Here is one way to do this.

- Design the program without paying much attention to the UI. Introduce as many classes as necessary to manage the data and computations that you need.
- Identify the classes that must respond to events. Keep the number of such classes as small as possible. If necessary, revise the design to reduce this number.
- Introduce a new class (or, if necessary, classes) to handle all of the events. When this class receives an event, it passes it to the object that handles that event.

13.2 The Model-View-Controller Framework

This idea was formalized long ago, in early versions of Smalltalk, as the *Model-View-Controller* (MVC) framework. The program has three main classes:

- The *model* does all of the important computation. The model should not be aware of the existence of any other program components. It responds to requests to performa various calculations and to report the results.
- The *view* provides one or more ways of displaying the results of computations on the screen. In a simple program, the view could manage one window showing results. The view should not be aware of other program components.
- The *controller* handles events and tells the model and the view what to do.

This description is somewhat idealized. In practice, the view usually communicates directly with the model because this is easier and more efficient than negotiating *via* the controller.

MVC is often implemented as a *framework*. This means that the model, view, and controller are abstract base classes that specify channels of communication. A program is implemented by deriving concrete classes from the base classes.

The following program is an extremely simple example of the MVC framework. Here are abstract base classes for the model and the view.

(Complete code on web page.)

```
class Model
{
    public:
        virtual int finished () { return false; }
        virtual void update () = 0;
        virtual double report () = 0;
};

class View
{
    public:
        virtual int finished () { return false; }
        virtual void update (double observable) = 0;
};
```

The class `Controller` is concrete here although in a practical application it should be abstract.

```
class Controller
{
    public:
        Controller (Model *model, View *view);
        void run ();
    private:
        Model *model;
        View *view;
};
```

The constructor is given a model and a view. Its `run()` method simply updates the model and the view until one of them indicates that it is finished or — as a safety precaution! — a fixed number of cycles have been executed.

```
Controller::Controller (Model *model, View *view)
{
    Controller::model = model;
    Controller::view = view;
}

void Controller::run ()
```

```
{
    int cycles = 0;
    while ( ! (model->finished() || view->finished() ) && cycles++ < 100)
    {
        model->update();
        view->update(model->report());
    }
}
```

We now introduce some concrete models. First, the class `Heater` simulates heating a room.

```
class Heater : public Model
{
    public:
        Heater ();
        void update ();
        double report () { return room_temperature; }
    private:
        int on;
        double room_temperature;
        double desired_temperature;
        double heater_temperature;
        double tolerance;
        double outside_temperature;
};
```

Here is the constructor for `Heater`.

```
Heater::Heater ()
{
    on = 0;
    room_temperature = 21.0;
    desired_temperature = 20.0;
    heater_temperature = 100.0;
    tolerance = 1.0;
    outside_temperature = -5.0;
}
```

The `update()` method for `Heater` tries to keep the room at the required temperature within the given tolerance.

```
void Heater::update ()
{
    room_temperature += 0.05 * (outside_temperature - room_temperature);
    if (on)
        room_temperature += 0.05 * (heater_temperature - room_temperature);
    if (on && room_temperature > desired_temperature + tolerance)
        on = false;
```

```
    if ( ! on && room_temperature < desired_temperature - tolerance)
        on = true;
}
```

The second model is a projectile that goes vertically upwards and comes down under the influence of gravity [Newton 1680].

```
class Projectile : public Model
{
    public:
        Projectile ();
        int finished ();
        void update ();
        double report ();
    private:
        double delta_t;
        double gravitational_constant;
        double velocity;
        double height;
};
```

The constructor sets initial values: the projectile is on the ground and ascending. The simulation finishes when the projectile lands on the ground.

```
Projectile::Projectile ()
{
    delta_t = 0.1;
    gravitational_constant = -5.0;
    velocity = 10.0;
    height = 0.0;
}

int Projectile::finished ()
{
    return height < 0.0;
}
```

The update function for the projectile uses Newton's laws.

```
void Projectile::update ()
{
    velocity += gravitational_constant * delta_t;
    height += velocity * delta_t;
}

double Projectile::report ()
{
    return height;
}
```

Next, we define concrete classes for views. The first is a "digital view": it simply displays a list of numbers.

```
class DigitalView : public View
{
    public:
        DigitalView ();
        int finished ();
        void update (double observable);
    private:
    int cycles;
};
```

The methods of a `DigitalView` are implemented as follows.

```
DigitalView::DigitalView ()
{
    cycles = 0;
}

int DigitalView::finished ()
{
    return cycles > 40;
}

void DigitalView::update (double observable)
{
    cout << observable << endl;
    cycles++;
}
```

The second kind of view is a very simple "graphical view" that displays indented asterisks.

```
class GraphicalView : public View
{
    public:
        void update (double observable);
};

void GraphicalView::update (double observable)
{
    const int num_stars = int(observable);
    for (int stars = 0; stars < num_stars; stars++)
        cout << '*';
    cout << endl;
}
```

The following main program does not follow the rules suggested above but could be made to do so in an appropriate environment.

```
int main ()
{
    Model *model;
    View *view;
    char reply;

    // Let the user choose a model.
    cout << "Do you want a (h)eater or a (p)rojectile? ";
    cin >> reply;
    switch (reply)
    {
        case 'h':
            model = new Heater;
            break;
        case 'p':
            model = new Projectile;
            break;
        default:
            model = new Heater;
            cout << "You got a heater.\n";
        break;
    }

    // Let the user choose a view.
    cout << "Do you want a (d)igital view or a (g)raphical view? ";
    cin >> reply;
    switch (reply)
    {
        case 'd':
            view = new DigitalView;
            break;
        case 'g':
            view = new GraphicalView;
            break;
        default:
            view = new DigitalView;
            cout << "You got a digital view.\n";
            break;
    }

    Controller ctrl(model, view);
    ctrl.run();
    return 1;
}
```